Senior iOS Developer Swift Interview Questions (2025 Edition)

1. Differences between `class`, `struct`, and `actor` in Swift

`class` (Reference Type)

- **Stored in Heap**: Objects live in memory until there are no references left.
- **Pass-by-Reference**: When assigned to a new variable, only the reference (pointer) is copied, not the data.
- **Supports Inheritance**: Classes can inherit properties and methods from another class.
- **Mutable Even If Declared with `let`**: Unlike structs, properties of a class can be modified even if the instance is assigned to a constant.

Example:

```
class Person {
    var name: String
    init(name: String) {
        self.name = name
    }
}

let person1 = Person(name: "Alice")
let person2 = person1 // Both person1 and person2 refer to the same memory address
person2.name = "Bob"

print(person1.name) // "Bob" (Changes reflect in both instances)
```

`struct` (Value Type)

- **Stored in Stack**: Structs are destroyed when they go out of scope, making them more memory efficient.
- **Pass-by-Value**: Assigning a struct to a new variable copies the data.
- **No Inheritance**: Structs do not support inheritance.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

- **Mutability Control**: Requires `mutating` keyword for modifying properties in methods.

Example:

```
struct Person {
    var name: String
}

var person1 = Person(name: "Alice")
var person2 = person1 // Creates a separate copy

person2.name = "Bob"
print(person1.name) // "Alice" (No change in the original instance)
```

`actor` (Reference Type with Concurrency)

- **Ensures Thread Safety**: Protects data from race conditions.
- **Sequential Access**: Allows only one task to access properties at a time.

Example:

```
actor BankAccount {
    private var balance: Int = 0

    func deposit(amount: Int) {
        balance += amount
    }

    func getBalance() -> Int {
        return balance
    }
}

let account = BankAccount()
Task {
    await account.deposit(amount: 100)
    print(await account.getBalance()) // Ensures safe access
```

```
}
```

---

2. Property Wrappers in Swift

Property wrappers allow reusable logic for properties.

Creating a Custom Property Wrapper

```
@propertyWrapper
struct Capitalized {
    private var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.capitalized }
    }
}

struct User {
    @Capitalized var name: String
}

var user = User(name: "john doe")
print(user.name) // "John Doe"
```

- **Use Case:** Automatically capitalize user input.

---

3. `@autoclosure` and Its Use

`@autoclosure` allows expressions to be automatically converted into closures.

Without `@autoclosure`

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

```
func logIfTrue(_ condition: () -> Bool) {
    if condition() {
        print("Condition met")
    }
}

logIfTrue({ 5 > 3 }) // Explicit closure
```

With `@autoclosure`

```
func logIfTrue(_ condition: @autoclosure () -> Bool) {
    if condition() {
        print("Condition met")
    }
}

logIfTrue(5 > 3) // No need to wrap in `{ }`
```

- **Use Case:** Avoid unnecessary computation (e.g., assertions, logging).

---

4. Swift Memory Management & ARC

- **Automatic Reference Counting (ARC)** tracks references to class instances.
- Objects are automatically **deallocated when reference count = 0**.

Example:

```
class Car {
    var model: String
    init(model: String) {
        self.model = model
        print("\(model) initialized")
    }
```

```
    deinit {
        print("\(model) deallocated")
    }
}

var car1: Car? = Car(model: "Tesla")
car1 = nil // "Tesla deallocated"
```

- **Use Case:** Efficient memory handling in Swift.

---

5. Retain Cycles & Prevention

A **retain cycle** happens when two objects strongly reference each other.

Retain Cycle Example

```
class Person {
    var name: String
    var pet: Pet? // Strong reference
    init(name: String) { self.name = name }
}

class Pet {
    var name: String
    var owner: Person? // Strong reference
    init(name: String) { self.name = name }
}
```

Solution: Use `weak` or `unowned`

```
class Person {
    var name: String
    weak var pet: Pet? // Weak reference prevents retain cycle
}
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

---

6. `inout` Parameters in Swift

- Allows modification of function parameters directly.

```
func doubleValue(_ number: inout Int) {
    number *= 2
}

var value = 10
doubleValue(&value)
print(value) // 20
```

🔹 **Use Case:** Modifying variables without returning a new value.

---

7. `Codable` & `Decodable` with Custom JSON Handling

Swift can encode and decode JSON using `Codable`.

Basic Example

```
struct User: Codable {
    var name: String
    var age: Int
}
```

Custom Decoding Example

```
struct User: Codable {
    var name: String
    var age: Int
```

```
    enum CodingKeys: String, CodingKey {
        case name
        case age = "user_age" // Maps JSON key to a different property
name
    }
}
```

- **Use Case:** Handling different JSON formats.

---

8. Escaping vs Non-Escaping Closures

Non-Escaping (Default)

```
func performTask(task: () -> Void) {
    task() // Executed immediately
}
```

Escaping (`@escaping`)

```
func performAsyncTask(completion: @escaping () -> Void) {
    DispatchQueue.global().async {
        sleep(2)
        completion()
    }
}
```

- **Use Case:** Asynchronous operations.

---

9. Function Builders in Swift

Used in SwiftUI to build complex structures.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

```swift
@resultBuilder
struct StringBuilder {
    static func buildBlock(_ components: String...) -> String {
        components.joined(separator: " ")
    }
}

func makeSentence(@StringBuilder _ content: () -> String) -> String {
    content()
}

let sentence = makeSentence {
    "Swift"
    "is"
    "awesome!"
}

print(sentence) // "Swift is awesome!"
```

- **Use Case:** DSLs like SwiftUI.

---

10. `Any` vs `AnyObject` vs `AnyHashable`

- **`Any`**: Any type (class, struct, enum, etc.).
- **`AnyObject`**: Only class types.
- **`AnyHashable`**: Any type that conforms to `Hashable`.

```swift
let anything: Any = 42
let anyObject: AnyObject = NSString("Hello")
let hashable: AnyHashable = "Swift"
```

- **Use Case:** Flexible APIs that accept multiple types.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

11. Difference Between `protocol` and `class` Inheritance

Class Inheritance

- **Single Inheritance**: A class can inherit from only one superclass.
- **State Management**: Can have stored properties.
- **Reference Type**: Modifying an instance affects all references.
- **Dynamic Dispatch**: Methods can be overridden using `override` and dynamically dispatched at runtime.

Example:

```
class Animal {
    var name: String
    init(name: String) { self.name = name }

    func makeSound() {
        print("Some generic animal sound")
    }
}

class Dog: Animal {
    override func makeSound() {
        print("Woof!")
    }
}

let dog = Dog(name: "Buddy")
dog.makeSound() // "Woof!"
```

Protocol Inheritance

- **Multiple Inheritance**: A type can conform to multiple protocols.
- **No Stored Properties**: Only defines behavior.
- **Static Dispatch**: Methods from protocols are usually resolved at compile time.

Example:

```swift
protocol Flyable {
    func fly()
}

protocol Swimmable {
    func swim()
}

class Duck: Flyable, Swimmable {
    func fly() { print("Duck is flying") }
    func swim() { print("Duck is swimming") }
}
```

🔹 **Key Takeaway**:

- Use **class inheritance** for **shared state**.
- Use **protocols** for **behavior composition**.

---

12. Associated Types in Swift Protocols

- `associatedtype` allows a protocol to **define a placeholder type**.
- The actual type is **determined by the conforming type**.

Example

```swift
protocol Container {
    associatedtype Item
    func add(_ item: Item)
    func getAll() -> [Item]
}

class IntContainer: Container {
    private var items: [Int] = []

    func add(_ item: Int) { items.append(item) }
    func getAll() -> [Int] { items }
```

```
}
```

🔹 **Why?**

- Increases flexibility.
- Allows defining **generic-like** behavior inside protocols.

---

13. Protocol-Oriented Programming (POP) vs Object-Oriented Programming (OOP)

OOP Approach

- Uses **inheritance** to share behavior.
- Often results in **deep class hierarchies**.

Example

```
class Animal {
    func makeSound() { print("Some animal sound") }
}

class Dog: Animal {
    override func makeSound() { print("Bark!") }
}
```

POP Approach

- Uses **protocol extensions** to share behavior.
- Encourages **composition over inheritance**.

Example

```
protocol SoundMaking {
    func makeSound()
}
```

```
extension SoundMaking {
    func makeSound() { print("Default sound") }
}

struct Dog: SoundMaking {
    func makeSound() { print("Bark!") }
}
```

- **POP avoids deep hierarchies, making code more modular and flexible.**

---

14. Purpose of `Self` in Protocols

- `Self` refers to the **conforming type** inside a protocol.

Example: Returning `Self`

```
protocol Cloneable {
    func clone() -> Self
}

class Car: Cloneable {
    func clone() -> Self { return self }
}
```

- **Why?** Ensures that `clone()` always returns the correct type.

---

15. Generic Constraints in Swift

- Constrain a generic type to **a specific requirement**.

Example: Generic with a Constraint

```
func findLargest<T: Comparable>(in array: [T]) -> T? {
    return array.max()
}

print(findLargest(in: [1, 2, 3, 4])) // 4
```

Using `where` Clause

```
func findElement<T>(in array: [T]) where T: Equatable {
    // Function body
}
```

- **Why?** Ensures type safety while allowing flexibility.

---

16. Extending a Protocol

- Protocol extensions allow adding **default implementations**.

Example

```
protocol Greetable {
    func greet()
}

extension Greetable {
    func greet() {
        print("Hello, World!")
    }
}

struct Person: Greetable {}

let person = Person()
person.greet() // "Hello, World!"
```

🔹 **Reduces boilerplate code** by providing default implementations.

---

17. Opaque Return Types (`some`) in Swift

- **Hides concrete return types** while ensuring type safety.

Example

```
protocol Shape {
    func area() -> Double
}

struct Circle: Shape {
    var radius: Double
    func area() -> Double { return .pi * radius * radius }
}

func getShape() -> some Shape {
    return Circle(radius: 5)
}
```

🔹 **Why?**

- Prevents exposing implementation details.
- Makes return types more flexible.

---

18. Covariance and Contravariance in Swift Generics

Covariance (Subtype Allowed)

- **Allows a subtype** where a supertype is expected.

```
class Animal {}
class Dog: Animal {}
```

```
let animals: [Animal] = [Dog()] // ✅ Allowed (Array is covariant)
```

Contravariance (Supertype Allowed)

- **Allows a supertype** where a subtype is expected.

```
func printAnimalInfo(_ animal: Animal) { print("Animal info") }
let dogPrinter: (Dog) -> Void = printAnimalInfo // ✅ Allowed
(Contravariant)
```

🔹 **Why?** Helps maintain **type safety**.

---

19. Handling Multiple Protocol Conformances with Conflicting Methods

When two protocols define the **same method**, Swift requires **explicit disambiguation**.

Example

```
protocol A {
    func greet()
}

protocol B {
    func greet()
}

struct MyStruct: A, B {
    func greet() { print("Hello from MyStruct") }
}
```

Solution: Use Extensions

```
extension MyStruct: A {
```

```
    func greet() { print("Hello from A") }
}

extension MyStruct: B {
    func greet() { print("Hello from B") }
}
```

- **Why?** Ensures correct method resolution.

---

20. `@objc` and Why It's Needed

- Used to expose Swift methods to **Objective-C runtime**.
- Required for **Selector-based APIs** (e.g., `#selector()`).

Example

```
import Foundation

class MyClass: NSObject {
    @objc func sayHello() {
        print("Hello, Objective-C!")
    }
}

let selector = #selector(MyClass.sayHello)
```

- **Why?**
  - Enables compatibility with **Objective-C frameworks**.
  - Allows method calls using `#selector()`.

---

21. Explain the difference between Grand Central Dispatch (GCD) and Operation Queues.

Grand Central Dispatch (GCD) and **Operation Queues (NSOperationQueue)** are both concurrency mechanisms in iOS, but they have key differences in flexibility and complexity:

Feature	GCD (`DispatchQueue`)	Operation Queue (`NSOperationQueue`)
Abstraction Level	Low-level API (C-based)	High-level API (Objective-C-based)
Task Representation	Uses closures (`DispatchWorkItem`)	Uses `NSOperation` subclasses
Dependency Handling	No built-in dependency management	Supports dependencies between operations
Task Cancellation	No direct cancellation (must handle manually)	Supports `cancel()` method for `NSOperation`
Priority Control	Uses `QoS` (Quality of Service)	More advanced priority control with dependencies
Observability	No direct KVO support	Supports KVO (Key-Value Observing)
Thread Safety	Basic synchronization tools	More robust with dependencies and execution states

👉 **Use GCD** when you need simple, lightweight, and low-overhead concurrency.
👉 **Use Operation Queues** when you need dependency management, cancellation, and observability.

---

22. What are the differences between `async`/`await`, `DispatchQueue`, and `NSOperationQueue`?

Feature	`async`/`await`	`DispatchQueue` (GCD)	`NSOperationQueue`
Abstraction Level	High-level (Swift concurrency)	Mid-level (GCD API)	High-level (Foundation API)
Concurrency Model	Structured concurrency	Task-based, dispatch queues	Object-oriented, operation queues

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

Syntax	`async let`, `await`	`DispatchQueue.async {}`	`OperationQueue.addOperation {}`
Cancellation	Supports cooperative cancellation (`Task.cancel()`)	No built-in cancellation	Supports `cancel()` on `NSOperation`
Dependency Management	Not directly supported (managed by `TaskGroup`)	No dependencies	Supports dependencies via `addDependency()`
Performance	Optimized for Swift's structured concurrency	Lightweight, efficient for simple tasks	More overhead due to object-oriented design

👉 **Use `async`/`await` for structured, safe concurrency with cooperative cancellation.**
👉 **Use GCD (`DispatchQueue`) for simple concurrency without dependencies.**
👉 **Use `NSOperationQueue` for complex dependencies and task management.**

---

23. What is an actor in Swift? How does it help in thread safety?

Actors in Swift are a concurrency-safe type that serializes access to its mutable state, preventing race conditions.

```
actor BankAccount {
    private var balance: Int = 0

    func deposit(amount: Int) {
        balance += amount
    }

    func getBalance() -> Int {
        return balance
    }
}
```

How Actors Ensure Thread Safety

1. **Encapsulation of State** – All properties inside an actor are protected from data races.

Asynchronous Access – Methods inside an actor must be accessed using `await`:

```
let account = BankAccount()
await account.deposit(amount: 100)
```

2.
3. **Automatic Serialization** – Multiple tasks calling an actor will be serialized, preventing simultaneous writes.

👉 Use actors when you need **safe concurrent access** to shared mutable state.

---

24. Explain `Task`, `TaskGroup`, and `DetachedTask` in Swift concurrency.

1. `Task` **(Structured Concurrency)**
    - Runs an asynchronous operation within the structured concurrency model.
    - Can inherit task priorities.

Example:

```
Task {
    await fetchData()
}
```

    -
2. `TaskGroup` **(Parallel Execution)**
    - Enables concurrent child tasks within a single parent scope.

Example:

```
await withTaskGroup(of: Int.self) { group in
    for i in 1...5 {
        group.addTask { return i * i }
    }
```

```
}
```

-
3. `DetachedTask` **(Unstructured Concurrency)**
    - Runs independently without inheriting priority or task context.

Example:

```
let handle = Task.detached {
    return await fetchData()
}
let result = await handle.value
```

-

👉 **Use `Task` for structured concurrency, `TaskGroup` for parallel execution, and `DetachedTask` when detachment from the parent context is required.**

---

25. How does `MainActor` work? When should you use it?

`@MainActor` ensures that code runs on the main thread.

Example

```
@MainActor
class ViewModel {
    var title: String = "Hello"

    func updateTitle() {
        title = "Updated"
    }
}
```

Or for a function:

```
@MainActor
```

```
func updateUI() {
    label.text = "Updated"
}
```

👉 **Use `MainActor` for UI updates and main-thread-bound tasks.**

---

26. What are structured and unstructured concurrency in Swift?

Type	Definition	Example
Structured Concurrency	Tasks are managed within a scope, ensuring they complete before the function exits.	`Task { await fetchData() }`
Unstructured Concurrency	Tasks run independently without direct supervision.	`Task.detached { await fetchData() }`

👉 Use **structured concurrency** (`Task`, `TaskGroup`) for safety, and **unstructured concurrency** (`DetachedTask`) when needed.

---

27. Explain the differences between `Task.sleep()` and `Thread.sleep()`.

Feature	`Task.sleep()`	`Thread.sleep()`
Concurrency Model	Works with Swift concurrency	Blocks the entire thread
Performance	Non-blocking, suspends only the task	Blocking, inefficient for concurrency
Usage	`await Task.sleep(1_000_000_000)`	`Thread.sleep(forTimeInterval: 1.0)`

👉 **Use `Task.sleep()` in Swift concurrency to avoid blocking threads.**

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

28. How does Swift handle race conditions?

Swift prevents race conditions using:

1. **Actors** (`@MainActor`, `actor`).
2. **Isolation Mechanisms** (`Task`, `TaskGroup`).
3. **Serial Dispatch Queues** (`DispatchQueue.serial`).
4. **Locks & Semaphores** (if needed in legacy code).

---

29. What is deadlock, and how can you prevent it?

A **deadlock** happens when two threads wait indefinitely for each other to release resources.

Example of Deadlock

```
let queue = DispatchQueue(label: "com.deadlock")
queue.sync {
    queue.sync { print("Deadlock") } // This will cause a deadlock
}
```

How to Prevent Deadlocks

1. **Avoid nested `sync` calls.**
2. **Use `async` instead of `sync` when dispatching tasks.**
3. **Use timeout mechanisms in locks.**

---

30. How would you handle a network request that requires multiple dependent API calls asynchronously?

Use **async/await** with structured concurrency:

```
func fetchUserProfile() async throws -> UserProfile {
    let user = try await fetchUser()
    let posts = try await fetchPosts(user.id)
    let friends = try await fetchFriends(user.id)

    return UserProfile(user: user, posts: posts, friends: friends)
```

```
}
```

👉 **Use `async/await` for dependencies and `TaskGroup` for parallel execution.**

---

31. Differences Between MVC, MVVM, and VIPER

These are architectural patterns used to organize iOS applications:

Model-View-Controller (MVC)

- **Structure:**
  - Model: Manages data and business logic.
  - View: Handles UI representation.
  - Controller: Acts as a mediator between Model and View.
- **Pros:** Simple and widely used.
- **Cons:** Leads to "Massive View Controller" due to bloated logic.
- **Use Case:** Best for small apps or when quick development is needed.

Model-View-ViewModel (MVVM)

- **Structure:**
  - Model: Data and logic.
  - View: UI representation.
  - ViewModel: Transforms data for the View and handles presentation logic.
- **Pros:** Better separation of concerns, improved testability.
- **Cons:** Requires more setup than MVC.
- **Use Case:** Best for medium to large apps, especially when using SwiftUI (because of bindings).

View-Interactor-Presenter-Entity-Router (VIPER)

- **Structure:**
  - View: UI Layer.
  - Interactor: Handles business logic.
  - Presenter: Prepares data for View.
  - Entity: Model layer.
  - Router: Handles navigation.
- **Pros:** High modularity, excellent testability.
- **Cons:** Complex and requires more code.
- **Use Case:** Large-scale applications with multiple layers of logic.

Choosing Between Them

- **MVC** → Small projects, fast prototyping.
- **MVVM** → Medium/large apps, better maintainability.
- **VIPER** → Large enterprise apps where modularity and scalability are key.

---

32. Using `Combine` for Reactive Programming

`Combine` is Apple's reactive framework that allows declarative and functional programming.

Example: Observing API Response

```
import Combine

struct APIClient {
    func fetchData() -> AnyPublisher<String, Error> {
        let url = URL(string: "https://api.example.com/data")!
        return URLSession.shared.dataTaskPublisher(for: url)
            .map { String(decoding: $0.data, as: UTF8.self) }
            .mapError { $0 as Error }
            .eraseToAnyPublisher()
    }
}
```

Using Combine with SwiftUI

```
class ViewModel: ObservableObject {
    @Published var data: String = ""
    var cancellables = Set<AnyCancellable>()

    func loadData() {
        APIClient().fetchData()
            .receive(on: DispatchQueue.main)
            .sink(receiveCompletion: { _ in }, receiveValue: {
self.data = $0 })
            .store(in: &cancellables)
    }
}
```

Use Cases

- Handling API responses.
- Real-time UI updates (binding ViewModel to UI).
- Chaining multiple asynchronous operations.

---

33. `ObservableObject`, `@Published`, and `@State` in SwiftUI

These are used to handle state management.

`ObservableObject`

A class that conforms to `ObservableObject` can notify SwiftUI views about state changes.

```
class ViewModel: ObservableObject {
    @Published var count = 0
}
```

`@Published`

Used within `ObservableObject` to notify when a property changes.

```
class ViewModel: ObservableObject {
    @Published var username: String = ""
}
```

`@State`

Used for local state within a SwiftUI View.

```
struct CounterView: View {
    @State private var count = 0
}
```

Differences

Property	Used in	Scope
`@State`	View	Local
`@Published`	ViewModel	Used with `ObservableObject`
`ObservableObject`	ViewModel	Global state

---

34. Dependency Injection in Swift

Dependency Injection (DI) allows passing dependencies instead of hardcoding them.

1. Constructor Injection

```
class NetworkService {
    func fetchData() { }
}

class ViewModel {
    let service: NetworkService
    init(service: NetworkService) {
        self.service = service
    }
}
```

2. Property Injection

```
class ViewModel {
    var service: NetworkService?
}
```

3. Factory Pattern

```
class DependencyFactory {
    static func createService() -> NetworkService {
```

```
        return NetworkService()
    }
}
```

4. Using Swinject (DI Framework)

```
let container = Container()
container.register(NetworkService.self) { _ in NetworkService() }
```

Choosing the Right Approach

- **Constructor Injection:** Preferred for immutability.
- **Property Injection:** Used when dependency can change.
- **Factory Pattern:** Best for centralized dependency management.
- **DI Frameworks:** Suitable for large applications.

---

35. Implementing Deep Linking in iOS

Deep linking allows opening specific app screens via URLs.

Using URL Schemes

1. Add a URL Scheme in `Info.plist`.
2. Handle in `AppDelegate`:

```
func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey: Any] = [:]) -> Bool {
    print("Opened with URL: \(url)")
    return true
}
```

Using Universal Links

1. Configure Associated Domains in `Entitlements.plist`.
2. Add an `apple-app-site-association` (AASA) file on the server.

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

36. Core Data vs Realm

Feature	Core Data	Realm
Apple-native	✅	❌
Performance	Good	Faster
Schema Migration	Complex	Easier
Multi-threading	Difficult	Easier
Relationship Handling	Complex	Simple

When to Choose?

- **Core Data:** When integrating with Apple's ecosystem.
- **Realm:** When performance and ease of use are priorities.

---

37. Diffable Data Sources

`DiffableDataSource` simplifies handling updates in `UITableView`/`UICollectionView`.

Traditional `UITableViewDataSource`

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    // Standard cell creation
}
```

Diffable Data Source

```
let dataSource = UITableViewDiffableDataSource<Section,
Item>(tableView: tableView) { tableView, indexPath, item in
    let cell = tableView.dequeueReusableCell(withIdentifier: "cell",
for: indexPath)
    cell.textLabel?.text = item.title
    return cell
}
```

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

Advantages

- Automatic animations.
- No need to manually call `reloadData()`.

---

38. Optimizing Scrolling Performance

- Use `cellForRowAt` efficiently.
- Implement cell reuse (`dequeueReusableCell`).
- Use lightweight views (`drawRect` sparingly).
- Enable prefetching (`UITableViewDataSourcePrefetching`).
- Use background threads for data processing.

---

39. Integrating SwiftUI with UIKit

Embedding SwiftUI in UIKit

```
let swiftUIView = UIHostingController(rootView: MySwiftUIView())
```

Embedding UIKit in SwiftUI

```
struct MyUIKitView: UIViewControllerRepresentable {
    func makeUIViewController(context: Context) -> UIViewController {
        return MyViewController()
    }
}
```

---

40. What are App Clips, and how do you implement them?

Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

App Clips are lightweight, fast, and focused mini versions of an iOS app that allow users to access specific app functionalities **without installing the full app**. They are designed for quick interactions (under 10MB) and can be launched via:

- **QR codes**
- **NFC tags**
- **Safari smart banners**
- **Messages and Maps links**
- **App Clip Codes (Apple-designed visual codes)**

🔹 **Example Use Cases:**

- Ordering food without downloading a restaurant's app
- Paying for parking
- Renting bikes or scooters
- Checking into a hotel

How to Implement App Clips?

1. Create an App Clip Target in Xcode

1. Open your main Xcode project.
2. Go to **File > New > Target**.
3. Select **App Clip** and add it to your existing app.

This creates a lightweight version of your app with a separate bundle identifier.

---

2. Design a Lightweight UI & Functionality

- Keep it under **10MB**.
- Load essential assets only.
- Use **on-demand resources** for extra assets if needed.
- Limit the app clip's purpose to a single task.

---

3. Implement App Clip Invocation Using NSUserActivity

In the App Clip's `SceneDelegate.swift`, handle the incoming URL:

```
func scene(_ scene: UIScene, continue userActivity:
NSUserActivity) {
```


Created By : Ruchit B Shah
(Senior Principal Software Engineer - Mobile Developer - 9228877722)

```
        guard let incomingURL = userActivity.webpageURL else { return
    }

        handleIncomingURL(incomingURL)

    }
```

This allows App Clips to recognize when launched via a URL or NFC tag.

---

4. Set Up Associated Domains

1. In **Signing & Capabilities**, add **Associated Domains** capability.
2. Include your website domain with `applinks:` and `appclip:` prefixes:

```
applinks:yourdomain.com

appclip:yourdomain.com
```

3. Deploy an **Apple App Site Association (AASA) file** at `https://yourdomain.com/.well-known/apple-app-site-association` with App Clip details.

Example AASA file:

```
{
  "applinks": {
    "apps": [],
    "details": [{
      "appID": "TEAM_ID.com.example.yourApp",
      "paths": ["/order/*"]
    }]
  },
```

```
  "appclips": {
    "apps": ["TEAM_ID.com.example.yourAppClip"]
  }
}
```

---

5. Add App Clip Experience in App Store Connect

1. Go to **App Store Connect > Your App**.
2. Navigate to **App Clip Experiences**.
3. Configure **Invocation Methods** (QR, NFC, URLs).
4. Associate it with a relevant **App Clip Card** (appears in Safari, Messages, etc.).

---

6. Test Your App Clip

- Use **Xcode's App Clip Invocation** feature.
- Scan the App Clip QR code in the developer menu.
- Test in Safari using `apple-app-site-association`.

---

41. What tools do you use for profiling and optimizing an iOS application?

Profiling and optimization in iOS are crucial to maintaining performance. The primary tools used are:

- **Xcode Instruments**: Helps analyze memory leaks, CPU usage, disk writes, network performance, etc.
- **Time Profiler**: Detects bottlenecks in CPU usage and function execution time.
- **Leaks Instrument**: Identifies memory leaks caused by retained objects.
- **Zombies Instrument**: Helps debug memory access issues by detecting messages sent to deallocated objects.
- **Energy Log**: Monitors energy impact, ensuring app efficiency.
- **Network Profiler**: Monitors API call response times and network usage.
- **Malloc and VM Tracker**: Analyzes memory allocation and deallocation patterns.
- **Static Analyzer**: Detects issues in the code before runtime.

---

42. How do you reduce an iOS app's memory footprint?

Reducing memory footprint ensures better performance and stability. Some best practices include:

- **Use ARC (Automatic Reference Counting)** to manage memory automatically.
- **Avoid retain cycles** by using weak/unowned references in closures and delegate patterns.
- **Efficient image handling**: Use `UIImage(named:)` instead of `UIImage(contentsOfFile:)` and prefer `ImageAssets`.
- **Use memory-efficient data structures** like `Set` and `Dictionary` instead of arrays where applicable.
- **Dispose of unused objects** by setting variables to `nil` when they are no longer needed.
- **Avoid overuse of singletons**, as they persist in memory throughout the app lifecycle.
- **Compress images** and **use WebP format** for better efficiency.
- **Release unused objects and caches** during memory warnings.

---

43. How does `Instruments` help in detecting memory leaks?

`Instruments` provides detailed memory analysis with these key features:

- **Leaks Instrument**: Detects memory leaks by tracking unreferenced objects that still exist in memory.
- **Allocations Instrument**: Monitors memory usage and highlights excessive allocation.
- **Zombies Instrument**: Detects messages sent to deallocated objects, preventing crashes.
- **Heapshot Analysis**: Captures memory states at different execution points, identifying objects that persist unexpectedly.
- **Retain Cycles Detection**: Finds objects that hold strong references to each other, preventing deallocation.

Using these tools, developers can identify and fix memory leaks, preventing performance degradation.

---

44. Explain lazy loading and its benefits in an iOS app.

Lazy loading is a technique where data or resources are loaded **only when needed** instead of during app startup.

Benefits:

- **Reduces initial memory consumption**, improving app launch speed.
- **Optimizes resource utilization**, loading heavy assets only when required.
- **Enhances scrolling performance** in lists (e.g., `UITableView` and `UICollectionView`) by loading images asynchronously.
- **Improves battery life** by reducing unnecessary processing.

Example: Lazy loading an image in `UITableViewCell`:

```
class CustomCell: UITableViewCell {
    var imageURL: URL? {
        didSet {
            loadImage()
        }
    }

    private func loadImage() {
        guard let url = imageURL else { return }
        DispatchQueue.global().async {
            if let data = try? Data(contentsOf: url),
               let image = UIImage(data: data) {
                DispatchQueue.main.async {
                    self.imageView?.image = image
                }
            }
        }
    }
}
```

---

45. What are the best practices for handling large images in an iOS app?

- **Use ImageAssets & Vector Graphics**: Prefer `SF Symbols` or `PDFs` over high-resolution raster images.
- **Use WebP Format**: Provides better compression than PNG or JPEG.
- **Resize Images Before Displaying**: Avoid storing oversized images in memory.
- **Load Images Lazily**: Use libraries like `SDWebImage` or `Kingfisher` for asynchronous loading and caching.

- **Use Core Animation & Metal**: Efficiently render images without blocking the main thread.
- **Avoid Using UIImage(contentsOfFile:)**: Use `UIImage(named:)` for caching benefits.

---

46. How do you optimize Core Data performance for large-scale applications?

- **Use Background Contexts**: Perform heavy operations in background threads.
- **Batch Inserts and Updates**: Minimize write operations using `NSBatchInsertRequest`.
- **Indexing**: Index frequently queried attributes to speed up fetch requests.
- **Faulting & Lazy Loading**: Avoid preloading all objects into memory.
- **Compact the SQLite Database**: Run `NSPersistentStoreCoordinator.execute(_:)` periodically.
- **Use Fetch Limits & Predicates**: Avoid fetching large datasets unnecessarily.

---

47. What are some strategies for improving app launch time?

- **Reduce App Bundle Size**: Remove unnecessary assets and use App Thinning.
- **Defer Non-Essential Work**: Load only critical components at startup.
- **Optimize Storyboards**: Avoid using a single massive storyboard.
- **Lazy Load Heavy Objects**: Defer database loading and networking calls.
- **Preload Data Efficiently**: Use background fetching and caching mechanisms.
- **Optimize Static Initializers**: Avoid expensive computations in `AppDelegate`.

---

48. Explain how to handle background execution and app lifecycle management.

iOS provides background execution modes:

- **Background Fetch**: Fetches new data periodically (`setMinimumBackgroundFetchInterval`).
- **Silent Push Notifications**: Updates app content without user interaction.
- **Background Tasks (BGTaskScheduler)**: Executes long-running tasks in the background.
- **Audio, Location, and VoIP Services**: Special cases where apps continue running.

Lifecycle Management:

- `applicationDidEnterBackground`: Save data and release resources.
- `applicationWillEnterForeground`: Restore the state and refresh UI.
- `applicationDidBecomeActive`: Restart paused tasks.
- `applicationWillTerminate`: Perform final cleanup.

---

49. What are some security best practices for storing sensitive data in an iOS app?

- **Use Keychain Services**: Store sensitive user credentials securely.
- **Encrypt Local Data**: Use **AES-256** for local file encryption.
- **Avoid Hardcoded Secrets**: Never store API keys or credentials in the app bundle.
- **Use Secure Enclaves**: Store biometric authentication data safely.
- **Enable App Transport Security (ATS)**: Force HTTPS connections.
- **Use Secure Storage**: Prefer `UserDefaults` only for non-sensitive data.
- **Implement Two-Factor Authentication (2FA)**: For better security.

---

50. How would you implement end-to-end encryption in an iOS app?

End-to-end encryption (E2EE) ensures that data remains encrypted from sender to receiver.

Steps to Implement:

1. **Generate Encryption Keys**: Use RSA (asymmetric) or AES (symmetric) encryption.

   Encrypt Data Before Sending:

   ```
   let key = SymmetricKey(size: .bits256)
   let sealedBox = try AES.GCM.seal(data, using: key)
   let encryptedData = sealedBox.combined
   ```

2. **Transmit Securely**: Use `HTTPS + TLS` with **Certificate Pinning**.

   Decrypt on the Receiver's End:

   ```
   let sealedBox = try AES.GCM.SealedBox(combined: encryptedData)
   let decryptedData = try AES.GCM.open(sealedBox, using: key)
   ```

3. **Use Secure Key Exchange**: Implement **Diffie-Hellman** or **Elliptic Curve Cryptography (ECC)**.
4. **Use Apple's Secure Enclave for Key Management**: Securely store private keys.