✅ Swift Language – iOS Interview Questions (with Answers & Code)

---

1. What are the key features of the Swift language?

- Modern and safe (type safety, optionals, null-safety).
- Fast (compiled with LLVM).
- Concise syntax (less boilerplate than Objective-C).
- Supports value & reference types.
- Protocol-oriented programming.
- Memory safe with ARC (Automatic Reference Counting).
- Interoperable with Objective-C.

---

2. What is a type-safe language and how does Swift enforce it?

- A **type-safe language** ensures variables only hold values of their declared type.
- Swift checks types **at compile-time**.

```
var age: Int = 25
age = "Hello" // ❌ Compile-time error
```

---

3. What is type inference in Swift?

- Swift infers the type of a variable if not explicitly declared.

```
let score = 100    // inferred as Int
let name = "DK"    // inferred as String
```

---

4. What is null safety in Swift and how are Optionals used to achieve it?

- Swift avoids `null` references using **Optionals (`?`)**.
- A variable can be `nil` only if it's optional.

```
var name: String? = nil  // Can be nil
var city: String = "Chennai" // Cannot be nil
```

---

5. What are Optionals in Swift and why are they important?

- **Optionals** represent a variable that may or may not contain a value.
- Important for avoiding runtime crashes like **NullPointerException** in other languages.

```
var email: String? = "test@example.com"
print(email ?? "No Email")  // Safe access
```

---

6. What is the difference between Optional Binding and Optional Chaining?

- **Optional Binding**: Safely unwrap using `if let` or `guard let`.
- **Optional Chaining**: Access properties/methods of an optional safely.

```
// Optional Binding
if let email = email {
    print("Email: \(email)")
}

// Optional Chaining
```

```
let length = email?.count  // returns Int?
```

---

7. What makes Enumerations powerful in Swift?

- Enums can have:
    - Associated values.
    - Raw values.
    - Methods and computed properties.
    - Conformance to protocols.

```
enum Result {
    case success(data: String)
    case failure(error: String)
}
```

---

8. What is the difference between let and var in Swift?

- `let`: constant (immutable).
- `var`: variable (mutable).

```
let pi = 3.14  // constant
var age = 25   // variable
```

---

9. Can a let constant reference a mutable object?

- Yes. `let` only makes the **reference immutable**, not the object itself.

```
class Person {
    var name: String
```

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

let person = Person(name: "DK")
person.name = "David"   // ✅ Allowed
// person = Person(name: "Alex") ❌ Not allowed (reference can't
change)
```

---

10. What happens when you declare a reference type using let?

- The reference cannot point to another instance,
- But the properties (if `var`) can still change.

---

11. How do you create read-only properties in Swift?

- Using `let` OR a computed property with `get` only.

```
class Circle {
    let radius: Double
    var area: Double { return 3.14 * radius * radius } // read-only
computed
    init(radius: Double) { self.radius = radius }
}
```

---

12. What is the difference between using let and a computed property with only a get?

- `let`: Value fixed at initialization.
- Computed property: Value dynamically calculated.

```
let fixedValue = 10  // constant
```

```
var dynamicValue: Int { return Int.random(in: 1...100) }
```

---

13. How do you make a property read-only outside the class but writable inside it?

- Use `private(set)`.

```
class Account {
    private(set) var balance = 0
    func deposit(_ amount: Int) {
        balance += amount
    }
}
```

---

14. How does Swift support both value and reference types?

- **Structs & Enums** → Value types (copied).
- **Classes** → Reference types (shared reference).

---

15. What is the difference between a value type and a reference type?

- **Value type**: Copy on assignment.
- **Reference type**: Shared reference.

```
struct Point { var x: Int }
var p1 = Point(x: 5)
var p2 = p1
p2.x = 10
print(p1.x) // 5 (copied)
```

---

16. What is the difference between a class and a struct in Swift?

Feature	Struct (Value type)	Class (Reference type)
Inheritance	❌ No	✅ Yes
Copy behavior	Copy on assignment	Shared reference
Memory	Stack	Heap
Mutability	Immutable if `let`	Properties can change

---

17. What are the practical use cases where a struct is preferred over a class?

- Lightweight models (`Point`, `Size`).
- When immutability is desired.
- When performance is important (stack-allocated).
- Used in SwiftUI `View`.

---

18. Why does SwiftUI use structs for defining views?

- **Value semantics** → easier to reason about.
- No side effects → UI updates predictable.
- Lightweight copies.

---

19. How is memory management handled in Swift?

- Swift uses **ARC (Automatic Reference Counting)**.

- Keeps track of references and deallocates objects when count = 0.

---

20. How is memory management different for classes and structs?

- **Classes (reference type)** → ARC manages reference counts.
- **Structs (value type)** → Stored on stack, automatically cleaned.

---

21. What is Copy-on-Write in Swift?

- **Optimization technique** for value types (Array, Dictionary, String).
- A copy of the data is only made when the **new copy is mutated**.

```
var arr1 = [1, 2, 3]
var arr2 = arr1   // arr2 shares same storage
arr2.append(4)    // Copy happens here (only on mutation)
```

---

22. How do Array, Dictionary, and String implement Copy-on-Write in Swift?

- They initially **share memory references**.
- When one changes, Swift creates a **separate copy** → prevents unintended mutations.

---

23. What is protocol-oriented programming and how is it different from object-oriented programming?

- **OOP**: Focuses on classes & inheritance.
- **POP**: Focuses on protocols & protocol extensions.
- Encourages **composition over inheritance**.

```
protocol Drivable {
    func drive()
}
extension Drivable {
    func drive() { print("Driving...") }
}
struct Car: Drivable {}
Car().drive()
```

---

24. How do Generics work in Swift and why are they useful?

- Generics allow writing **type-safe reusable code** without duplication.

```
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}
```

---

25. What are collection types in Swift?

- **Array**: Ordered collection.
- **Set**: Unordered unique elements.
- **Dictionary**: Key-value pairs.

---

26. What are the types of access control modifiers available in Swift?

- `open` → Accessible & overridable outside module.
- `public` → Accessible outside module, not overridable.

- `internal` (default) → Accessible within module.
- `fileprivate` → Accessible within file.
- `private` → Accessible within scope only.

---

27. Which access modifier is the default in Swift?

- **internal**

---

28. What is the difference between open and public access modifiers?

- `open`: Allows **subclassing & overriding** outside the module.
- `public`: Accessible, but **cannot be subclassed or overridden** outside module.

---

29. What is the difference between private and fileprivate in Swift?

- `private`: Scope-limited (within class/struct).
- `fileprivate`: Accessible anywhere in the **same file**.

---

30. How does the private(set) access modifier help in encapsulating data while allowing read-only access?

- Allows **read access** outside, **write access** only inside the type.

```
class Counter {
    private(set) var value = 0
    func increment() { value += 1 }
```

```
}
```

---

31. What are the key differences between stored properties and computed properties in Swift?

- **Stored property**: Holds actual value in memory.
- **Computed property**: Returns value dynamically (no storage).

```
struct Circle {
    var radius: Double
    var area: Double { 3.14 * radius * radius } // computed
}
```

---

32. What is the purpose of the lazy keyword in Swift, and in which scenarios is it best used?

- **Initialized only when accessed** (lazy loading).
- Useful for expensive operations.

```
class DataManager {
    lazy var data = loadData()  // expensive operation
    func loadData() -> [String] { ["A", "B", "C"] }
}
```

---

33. How does the inout keyword enable pass-by-reference behavior in Swift functions?

- Normally parameters are **passed by value**.
- `inout` allows modification inside a function to reflect outside.

```
func double(_ number: inout Int) {
```

```
    number *= 2
}
var n = 5
double(&n)
print(n) // 10
```

---

34. What is the role of the defer keyword in Swift, and how is it typically used for resource management?

- Executes a block **before function exits**, regardless of control flow.
- Useful for **cleanup** (closing files, releasing resources).

```
func readFile() {
    print("Open File")
    defer { print("Close File") }
    print("Read Data")
}
readFile()
// Output: Open File → Read Data → Close File
```

---

35. How is operator overloading implemented in Swift, and how does it contribute to expressive code?

- Define custom implementations of operators for your types.

```
struct Vector {
    var x, y: Int
    static func + (lhs: Vector, rhs: Vector) -> Vector {
        return Vector(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
    }
}
let v1 = Vector(x: 2, y: 3)
let v2 = Vector(x: 4, y: 5)
print(v1 + v2) // Vector(x:6,y:8)
```

---

36. What is the process to create a custom operator in Swift, and how do operator types like infix, prefix, and postfix affect its usage?

1. Declare the operator.
2. Implement function.

- **infix**: Between operands (`a + b`).
- **prefix**: Before operand (`-a`).
- **postfix**: After operand (`a!`).

```
prefix operator √
prefix func √ (value: Double) -> Double {
    return sqrt(value)
}
print(√9) // 3.0
```

---

37. How do extensions enhance the functionality of existing types in Swift, and what limitations exist when using them?

- Extensions allow **adding methods, computed properties, initializers** to existing types.
- Limitations:
    - Cannot add **stored properties**.
    - Cannot **override** existing methods.

```
extension Int {
    var squared: Int { return self * self }
}
print(5.squared) // 25
```

---

38. Why are stored properties not allowed inside Swift extensions?

- Extensions do not allocate new storage → only add **behavior**, not data.

---

39. Why can't you override methods inside an extension even if the class is subclassed?

- Extensions are meant for **adding functionality**, not modifying existing behavior.
- Overriding is reserved for subclassing.

---

40. How does Swift promote protocol-oriented programming and what benefits does it offer over traditional object-oriented programming?

- Promotes **protocols + protocol extensions** instead of inheritance.
- Benefits:
    - Code reusability.
    - Avoids deep inheritance hierarchies.
    - Multiple protocol conformance.

---

41. What makes protocol extensions in Swift a powerful tool for writing reusable and modular code?

- Provide **default method implementations**.
- Allow **shared behavior** without base classes.

```
protocol Drawable { func draw() }
```

```
extension Drawable {
    func draw() { print("Default Draw") }
}
struct Shape: Drawable {}
Shape().draw()  // Default Draw
```

---

42. How do associated types enable protocols to remain generic while being strongly typed in Swift?

- Allow **placeholder types** in protocols.

```
protocol Container {
    associatedtype Item
    func add(_ item: Item)
}
```

---

43. How does conditional conformance allow Swift types to adopt a protocol only under specific conditions?

- Types can conform to protocols **only when constraints are met**.

```
extension Array: Equatable where Element: Equatable {}
```

---

44. How do generics in Swift contribute to type-safe and reusable code across different data types?

- Ensure code works with **any type**, but still type-checked.

```
struct Stack<T> {
    private var items = [T]()
    mutating func push(_ item: T) { items.append(item) }
    mutating func pop() -> T { items.removeLast() }
}
```

---

45. What are some real-world scenarios where using generics in Swift significantly improves code reusability and maintainability?

- Collections (Array, Dictionary).
- Reusable data structures (Stack, Queue).
- Networking layers with `Result<T, Error>`.
- SwiftUI Views (`ForEach<T>`).

---

46. What is a subscript in Swift and how can you create a custom subscript in your type?

- Subscripts allow **accessing elements using `[ ]` syntax**.
- Useful in **collections, dictionaries, custom data structures**.

```
struct TimesTable {
    var multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}

let table = TimesTable(multiplier: 3)
print(table[4]) // 12
```

---

47. What is Codable in Swift and how does it relate to Encodable and Decodable?

- **Codable** = `Encodable + Decodable`.

- Used to convert Swift objects ↔ JSON / Plist.

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

---

48. Is Codable a class, struct, or protocol? How is it used in practice?

- `Codable` is a **protocol (typealias)**.
- Usage: With `JSONEncoder` and `JSONDecoder`.

```
let user = User(name: "DK", age: 30)
let jsonData = try JSONEncoder().encode(user)
let decoded = try JSONDecoder().decode(User.self, from: jsonData)
```

---

49. How do enumerations work in Swift, and what are associated values in enums?

- Enums define **finite set of cases**.
- Associated values let cases **store extra data**.

```
enum Result {
    case success(data: String)
    case failure(error: String)
}
```

---

50. How can you use associated values in enums to represent complex state?

- Useful for modeling multiple data cases.

```
enum NetworkState {
    case loading
    case success(data: String)
    case error(code: Int, message: String)
}
```

---

51. What is the Result type in Swift, and how does it use associated values to represent success or failure?

Built-in enum:

```
enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}
```

-
- Used in networking, async tasks.

```
func fetchData() -> Result<String, Error> {
    return .success("Data received")
}
```

---

52. What are the different types of initializers in Swift?

- **Designated initializer**: Primary initializer.
- **Convenience initializer**: Secondary helper initializer.
- **Failable initializer**: May return `nil`.
- **Required initializer**: Must be implemented by subclasses.

---

53. What is a convenience initializer, and what conditions must be met to use one?

- Provides **secondary initialization**.
- Must call a **designated initializer**.

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

---

54. What is a failable initializer in Swift, and when should you use it?

- Returns `nil` if initialization fails.
- Use when object creation may fail.

```
struct User {
    var age: Int
    init?(age: Int) {
        if age < 0 { return nil }
        self.age = age
    }
}
```

---

55. What are higher-order functions in Swift and how do they enable functional programming?

- Functions that **take functions as parameters** or **return functions**.
- Examples: `map`, `filter`, `reduce`.

---

56. How does the map function work in Swift collections?

- Transforms each element into a new form.

```
let numbers = [1, 2, 3]
let squares = numbers.map { $0 * $0 }
print(squares) // [1,4,9]
```

---

57. What is the purpose of compactMap, and how does it differ from map?

- `map`: Returns array with optionals.
- `compactMap`: Removes `nil` values.

```
let values = ["1", "a", "3"]
let result = values.compactMap { Int($0) }
print(result) // [1, 3]
```

---

58. How is flatMap used in Swift, and what makes it different from map and compactMap?

- Flattens nested collections.

```
let nested = [[1, 2], [3, 4]]
let flat = nested.flatMap { $0 }
print(flat) // [1,2,3,4]
```

---

59. What is the difference between map, compactMap, and flatMap in Swift?

Function	Purpose
`map`	Transforms each element (keeps optionals).
`compactMap`	Transforms + removes `nil`s.

`flatMap`	Flattens nested collections.

---

60. How does the sort function work in Swift, and how can you use custom sorting logic?

- `sort()` → in-place.
- `sorted()` → returns new array.

```
let arr = [3, 1, 2]
let sortedArr = arr.sorted(by: >) // descending
```

---

61. What is the reduce function in Swift and how can it be used to combine elements of a collection?

- Combines array elements into a single value.

```
let numbers = [1, 2, 3, 4]
let sum = numbers.reduce(0, +)
print(sum) // 10
```

---

62. What are the main collection types in Swift, and how do they differ?

- **Array**: Ordered, duplicates allowed.
- **Set**: Unordered, unique values.
- **Dictionary**: Key-value pairs.

---

63. How is an Array different from a Set in Swift?

- **Array**: Ordered, allows duplicates.

- **Set**: Unordered, unique values.

---

64. Why does Set in Swift only store unique values?

- Uses **hashing** to ensure uniqueness.
- Elements must conform to `Hashable`.

---

65. What are the key features and use-cases for Dictionary in Swift?

- **Key-value pairs**.
- Fast lookups (`O(1)` complexity).
- Use when you need **associative mapping**.

---

66. What is the role of the Hashable protocol in Swift, and when should a type conform to it?

- Required for **Set** and **Dictionary keys**.
- Ensures objects can be uniquely identified by a hash value.

```
struct Person: Hashable {
    let id: Int
}
```

---

67. How does Equatable enable comparison of values in Swift?

- Provides `==` operator for custom types.

```
struct Point: Equatable {
    var x, y: Int
}
print(Point(x:1,y:2) == Point(x:1,y:2)) // true
```

---

68. What is the Comparable protocol and how does it help in sorting custom types?

- Provides < operator → enables sorting.

```
struct Person: Comparable {
    var age: Int
    static func < (lhs: Person, rhs: Person) -> Bool {
        lhs.age < rhs.age
    }
}
```

---

69. What are the differences between closures and delegates in Swift, and when should each be used?

- **Closure**: Lightweight, inline callback.
- **Delegate**: Protocol-based, reusable, structured.

✅ Use closure → one-off callback.
✅ Use delegate → multiple responsibilities, reusable API.

---

70. What are Property Wrappers in Swift?

- A reusable way to define logic around properties.

```
@propertyWrapper
struct Clamped {
    private var value: Int = 0
```

```
    var wrappedValue: Int {
        get { value }
        set { value = min(max(newValue, 0), 100) }
    }
}
struct Player {
    @Clamped var score: Int
}
```

---

71. What are Opaque Types in Swift?

- Declared using `some`.
- Caller doesn't know exact type, only conforms to protocol.

```
func makeShape() -> some Equatable {
    return 5
}
```

---

72. What are some common keywords in Swift?

- `let`, `var`, `func`, `class`, `struct`, `enum`, `protocol`, `extension`,
- `guard`, `if`, `switch`, `defer`, `lazy`, `inout`, `associatedtype`,
- `some`, `throws`, `try`, `catch`, `async`, `await`.

📦 Local Data Storage in iOS

---

1. What is Core Data?

- Core Data is Apple's **object graph and persistence framework**.
- It manages object lifecycle, relationships, and persistence.
- Stores data in **SQLite, Binary, In-Memory, or XML (deprecated)** formats.

✅ Example:

```
let entity = NSEntityDescription.entity(forEntityName: "User", in:
context)!
let user = NSManagedObject(entity: entity, insertInto: context)
user.setValue("DK", forKey: "name")
try context.save()
```

---

2. What is Realm?

- Realm is a **third-party mobile database** (faster than Core Data in many cases).
- Uses its own persistence engine (not built on SQLite).
- Supports Swift/Kotlin cross-platform usage.

---

3. What is SQLite?

- **Lightweight relational database engine**.
- Can be used directly with SQL queries.
- Core Data itself uses SQLite under the hood (if chosen).

✅ Example (raw SQLite in Swift):

```
import SQLite3
// open DB, create tables, run queries
```

---

4. What are migrations in Core Data?

- Migration = Process of upgrading schema without losing existing data.
- Happens when you **add/remove/rename entities or attributes**.
- Core Data supports:
    - **Lightweight Migration** (automatic, when changes are simple).
    - **Manual Migration** (custom mapping for complex changes).

---

5. How to insert data using background queue in Core Data?

✅ Example:

```
persistentContainer.performBackgroundTask { context in
    let user = User(context: context)
    user.name = "Background Insert"
    try? context.save()
}
```

---

6. Difference between Core Data and SQLite?

Feature	Core Data	SQLite
Type	Object graph management	Relational DB
API	High-level (entities, relationships)	SQL queries
Relationships	Built-in	Manual (foreign keys)
Performance	Optimized caching, fetch requests	Raw queries, faster for large datasets

Encryption	Not built-in	Supported via SQLCipher

---

7. Is Core Data encrypted?

- ❌ By default, **Core Data is not encrypted**.
- ✅ You can encrypt using **File Protection APIs** or third-party libs (e.g., SQLCipher).

---

8. What are the delete rules in Core Data?

Delete rules define what happens to related objects when an object is deleted.

---

a) No Action

- Nothing happens to related objects.
- Risk: May leave orphan objects.

---

b) Nullify

- Related objects' relationship is set to `nil`.
- Object remains in store.

---

c) Cascade

- Deletes related objects automatically.
- Like SQL cascade delete.

---

d) Deny

- Prevents deletion if related objects exist.

---

9. Types of Persistent Store in Core Data

1. **SQLite Store** – Stores as SQLite DB (most common).
2. **Binary Store** – Stores in binary format (fast, but not human-readable).
3. **XML Store** – Stores as XML (deprecated, not recommended).
4. **In-Memory Store** – Stores only in RAM (good for temporary/cache data).

---

10. Concurrency Types in Core Data

- Used with `NSManagedObjectContext` for thread safety.

a) NSMainQueueConcurrencyType

- Runs on **main queue**.
- Used for **UI updates**.

b) NSPrivateQueueConcurrencyType

- Runs on **private background queue**.
- Used for **heavy operations** (inserts, fetches).

---

11. How to pass Managed Object between contexts on different queues?

- You cannot directly pass `NSManagedObject` across contexts.

- ✅ Instead, pass **NSManagedObjectID**:

```
let objectID = user.objectID
backgroundContext.perform {
    let user = backgroundContext.object(with: objectID)
}
```

---

12. Performance Guidelines for Core Data

- Use **batch operations** (`NSBatchInsertRequest`, `NSBatchDeleteRequest`).
- Use **faulting** and **fetch limits** to avoid loading all objects.
- Use **background contexts** for inserts/updates.
- Avoid passing large object graphs across threads.
- Use **Lightweight Migration** where possible.

🧬 App Lifecycle in iOS

---

1. Lifecycle of `AppDelegate` and `SceneDelegate`

Before iOS 13 (AppDelegate only)

- `application(_:didFinishLaunchingWithOptions:)` → App started.
- `applicationDidBecomeActive(_:)` → App is active.
- `applicationWillResignActive(_:)` → App will move to inactive (e.g., phone call).
- `applicationDidEnterBackground(_:)` → App moved to background.
- `applicationWillEnterForeground(_:)` → App is coming back to foreground.
- `applicationWillTerminate(_:)` → App is about to terminate.

iOS 13+ (AppDelegate + SceneDelegate)

- **AppDelegate**
    - Handles **app-wide events** (push notifications, background fetch, app launch).
- **SceneDelegate**
    - Handles **UI scene sessions** (multi-window support on iPad).

✅ Typical order (iOS 13+):

1. `application(_:didFinishLaunchingWithOptions:)`
2. `scene(_:willConnectTo:options:)`
3. `sceneDidBecomeActive(_:)`

4. `sceneWillResignActive(_:)`
5. `sceneDidEnterBackground(_:)`
6. `sceneWillEnterForeground(_:)`

---

2. Lifecycle of `UIViewController`

A `UIViewController` has well-defined lifecycle callbacks:

1. `init(coder:)` / `init(nibName:bundle:)` → Controller created.
2. `loadView()` → Creates view hierarchy.
3. `viewDidLoad()` → Called once, after view is loaded into memory.
4. `viewWillAppear(_:)` → Before view appears on screen.
5. `viewDidAppear(_:)` → After view appears on screen.
6. `viewWillDisappear(_:)` → Before view disappears.
7. `viewDidDisappear(_:)` → After view disappears.
8. `deinit` → Called when controller is deallocated.

✅ Example for debugging lifecycle:

```
override func viewDidLoad() {
    super.viewDidLoad()
    print("viewDidLoad")
}
```

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    print("viewWillAppear")
}
```

---

3. Lifecycle of `UIView`

A `UIView`'s lifecycle is tied to its creation, layout, and drawing:

1. `init(frame:)` / `init(coder:)` → View created.
2. `awakeFromNib()` → Called when view is loaded from storyboard/XIB.
3. `didMoveToSuperview()` → Called when added to a superview.
4. `didMoveToWindow()` → Called when added to a window.
5. `layoutSubviews()` → Called when layout changes (rotation, resizing).
6. `draw(_:)` → Called when the view is rendered (custom drawing).
7. `removeFromSuperview()` → Called when removed.
8. `deinit` → View deallocated.

✅ Example:

```
override func layoutSubviews() {
    super.layoutSubviews()
    print("layoutSubviews called")
}
```

📌 **Interview Tip**:

- For `AppDelegate/SceneDelegate`, stress the difference **pre/post iOS 13**.
- For `UIViewController`, emphasize that `viewDidLoad` is called **once**, but `viewWillAppear`/`viewDidAppear` can be called **many times**.
- For `UIView`, remember `layoutSubviews()` is called **often** → don't put heavy logic inside.

🌐 REST API Integration

---

1. What is URLSession in Swift?

- A class used to perform **network tasks** (data transfer, downloading, uploading).
- Supports **REST API calls** asynchronously.

```
let url = URL(string: "https://jsonplaceholder.typicode.com/posts")!
let task = URLSession.shared.dataTask(with: url) { data, response,
error in
    if let data = data {
        print(String(data: data, encoding: .utf8)!)
    }
}
task.resume()
```

---

2. What is URLRequest in Swift?

- Represents a **request** to a server with:
    - URL
    - HTTP method (GET, POST, PUT, DELETE)
    - Headers
    - Body

```
var request = URLRequest(url: URL(string:
"https://api.example.com/login")!)
request.httpMethod = "POST"
request.addValue("application/json", forHTTPHeaderField:
"Content-Type")
```

```
request.httpBody = try? JSONSerialization.data(withJSONObject:
["username": "dk", "password": "1234"])
```

---

3. What is URLComponents in Swift?

- Used to **safely construct URLs** with query parameters.

```
var components = URLComponents(string:
"https://api.example.com/search")!
components.queryItems = [
    URLQueryItem(name: "q", value: "swift"),
    URLQueryItem(name: "page", value: "1")
]
let url = components.url!
```

---

4. What are the different HTTP methods?

- **GET** → Fetch data
- **POST** → Create data
- **PUT** → Update entire resource
- **PATCH** → Update part of resource
- **DELETE** → Remove data
- **HEAD** → Fetch headers only

---

5. What are HTTP response status codes?

- **1xx** → Informational

- **2xx** → Success (`200 OK`, `201 Created`)
- **3xx** → Redirection (`301 Moved Permanently`)
- **4xx** → Client error (`400 Bad Request`, `401 Unauthorized`, `404 Not Found`)
- **5xx** → Server error (`500 Internal Server Error`, `503 Service Unavailable`)

---

6. What is OAuth protocol?

- A **secure authentication** protocol for APIs.
- Common flows:
    - **OAuth 2.0** with `Access Token` & `Refresh Token`.
- Used by Google, Facebook, GitHub APIs.

---

7. What is caching in iOS apps?

- Storing network responses/data locally to avoid repeated API calls.
- Done via:
    - `URLCache` (default)
    - Custom cache (e.g., saving JSON in Core Data/Realm).

```
let config = URLSessionConfiguration.default
config.requestCachePolicy = .returnCacheDataElseLoad
let session = URLSession(configuration: config)
```

---

8. What is Result type in Swift?

- A type-safe way to handle **success or failure**.

```
enum APIError: Error { case invalidResponse }

func fetchData(completion: @escaping (Result<String, APIError>) ->
Void) {
    completion(.success("Data fetched"))
    // or completion(.failure(.invalidResponse))
}
```

---

9. How do you handle retry calls in API requests?

- Use **exponential backoff** (retry after increasing delay).
- Can use libraries like **Alamofire** or custom logic:

```
func fetchWithRetry(url: URL, retries: Int = 3) {
    URLSession.shared.dataTask(with: url) { data, _, error in
        if let error = error, retries > 0 {
            print("Retrying... left: \(retries)")
            DispatchQueue.global().asyncAfter(deadline: .now() + 2) {
                fetchWithRetry(url: url, retries: retries - 1)
            }
        } else {
            print("Success!")
        }
    }.resume()
}
```

---

10. How do you check for internet availability in iOS?

- Since iOS 12+, use **NWPathMonitor** (Network framework).

```
import Network

let monitor = NWPathMonitor()
monitor.pathUpdateHandler = { path in
    if path.status == .satisfied {
        print("Internet available")
    } else {
        print("No internet")
    }
}
let queue = DispatchQueue(label: "NetworkMonitor")
monitor.start(queue: queue)
```

📌 **Interview Tip**

- Mention `URLSession` is **native** & lightweight, while `Alamofire` simplifies requests.
- Highlight **error handling** (timeouts, retries, JSON decoding).
- Emphasize **Result type** for modern Swift API handling.

🧠 Memory Management in Swift

---

1. What are Strong, Weak, and Unowned references in Swift?

- **Strong Reference (default)**
    - Increases the retain count of an object.
    - Keeps the object alive as long as at least one strong reference exists.

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

var person1: Person? = Person(name: "DK")  // strong ref
var person2 = person1                      // another strong ref
```

- **Weak Reference**
    - Does **not** increase retain count.
    - Can become `nil` if the object is deallocated.
    - Must always be declared as **optional**.

```
class Car {
    weak var owner: Person?   // weak to avoid retain cycle
}
```

- **Unowned Reference**
    - Does not increase retain count.

- Assumes the object will **never be nil** once set.
- Used in cases where the referenced object's lifetime is guaranteed.

```
class CreditCard {
    unowned let customer: Person   // never nil, crash if accessed
after deallocation
}
```

---

2. When should you use Weak and Unowned references in Swift?

- **Weak** → Use when the reference may become `nil` during its lifetime.
  - Example: Delegate patterns, parent-child relationships.
- **Unowned** → Use when the reference will always exist as long as this object exists.
  - Example: Two objects referencing each other where one has a shorter lifetime (CreditCard ↔ Customer).

---

3. What are the common ways to prevent memory leaks during development?

✅ Use **weak** or **unowned** in closures:

```
class ViewController: UIViewController {
    var name = "DK"
    func loadData() {
        fetchData { [weak self] data in   // prevents retain cycle
            self?.process(data)
        }
    }
}
```

✅ Break strong reference cycles:

- Between **objects** (`weak` delegates).
- Between **closures & self** (`[weak self]` capture list).

✅ Use **Instruments** → Leaks & Allocations to monitor memory.

---

4. How is the deinit method used in Swift?

- Called **just before an object is deallocated**.
- Useful for cleanup (remove observers, close connections).

```
class FileHandler {
    init() { print("File opened") }
    deinit { print("File closed") }
}

var file: FileHandler? = FileHandler()
file = nil   // triggers deinit → "File closed"
```

---

5. How can you find memory leaks, memory allocations, and retain cycles?

- **Xcode Debug Memory Graph**
    - Run app → `Debug Memory Graph` → shows retain cycles visually.
- **Instruments (Leaks & Allocations tool)**
    - Detects memory leaks, retain cycles, and excessive allocations.

- **Third-party tools** (less common in interviews):
    - Facebook's **FBRetainCycleDetector**.

⚙️ GCD and OperationQueue

---

1. What is the difference between GCD and OperationQueue?

- **GCD (Grand Central Dispatch)**
    - A **C-level API** for managing concurrency.
    - Lightweight, efficient, and best for **simple tasks**.
    - Provides queues (`serial` / `concurrent`) and async scheduling.
- **OperationQueue**
    - Built **on top of GCD** (Objective-C/Swift API).
    - Provides **higher-level abstraction**:
        - Dependencies between tasks.
        - Priorities.
        - Cancel operations.
        - KVO-compliant for monitoring.

👉 Use **GCD** for lightweight async code.
👉 Use **OperationQueue** when you need dependencies, cancellation, or progress tracking.

---

2. What are the different types of queues in GCD?

- **Main Queue**
    - Runs on the **main thread** → UI updates.

- **Global Queue**
    - Shared system-provided concurrent queues.
- **Custom Serial Queue**
    - Executes tasks one after another in order.
- **Custom Concurrent Queue**
    - Executes multiple tasks at the same time.

```
let serialQueue = DispatchQueue(label: "com.dk.serial")
let concurrentQueue = DispatchQueue(label: "com.dk.concurrent",
attributes: .concurrent)
```

---

3. What is the difference between Serial and Concurrent queues?

- **Serial Queue**
    - Executes **one task at a time** in FIFO order.
    - Useful when tasks must not overlap.
- **Concurrent Queue**
    - Executes **multiple tasks simultaneously**.
    - Order of **start is FIFO**, but **finish order is not guaranteed**.

---

4. What is Quality of Service (QoS) in GCD?

- Defines **priority & importance** of a task.

Types:

1. **.userInteractive** → Highest priority (UI updates, animations).
2. **.userInitiated** → Tasks requested by user, need quick results.
3. **.utility** → Long-running tasks (downloads, processing).
4. **.background** → Lowest priority (data sync, backups).

```
DispatchQueue.global(qos: .userInitiated).async {
    // quick user-requested task
}
```

---

5. What is DispatchWorkItem in GCD?

- A wrapper around a block of code that can be:
    - Executed.
    - Canceled.
    - Added to a queue.

```
let workItem = DispatchWorkItem {
    print("Task executed")
}
DispatchQueue.global().async(execute: workItem)
workItem.cancel()   // prevents execution if not started
```

---

6. What is DispatchGroup in GCD?

- Used to group multiple async tasks and get notified when **all complete**.

```
let group = DispatchGroup()

group.enter()
```

```
DispatchQueue.global().async {
    print("Task 1")
    group.leave()
}

group.enter()
DispatchQueue.global().async {
    print("Task 2")
    group.leave()
}

group.notify(queue: .main) {
    print("All tasks finished ✅")
}
```

---

7. What is BlockOperation in OperationQueue?

- A concrete subclass of `Operation`.
- Encapsulates one or more blocks of code.
- Supports cancellation and dependencies.

```
let blockOp = BlockOperation {
    print("Task 1")
}
blockOp.addExecutionBlock {
    print("Task 2")
}

let queue = OperationQueue()
queue.addOperation(blockOp)
```

👉 Unlike GCD, you can **add dependencies** between operations:

```
let op1 = BlockOperation { print("Op1") }
```

```
let op2 = BlockOperation { print("Op2") }
op2.addDependency(op1)
queue.addOperations([op1, op2], waitUntilFinished: false)
```

📌 **Interview Tip**:

- If they ask **“which to use?”** → Answer:
    - **GCD** for lightweight async code.
    - **OperationQueue** for task dependencies, cancellation, or progress.

🌀 Functional Reactive Programming – Combine

---

1. What are Publishers and Subscribers in Combine?

- **Publisher** → Emits values over time.
- **Subscriber** → Listens to values from a publisher and reacts.

```
let publisher = [1, 2, 3].publisher
let subscriber = publisher.sink { value in
    print("Received: \(value)")
}
```

---

2. What are the subjects in Combine? How do PassThroughSubject and CurrentValueSubject work?

- **Subjects** = special publishers you can manually send values to.
- **PassThroughSubject** → Starts empty, doesn't keep history.

```
let subject = PassthroughSubject<String, Never>()
subject.sink { print("Subscriber:", $0) }
subject.send("Hello") // emits
```

- **CurrentValueSubject** → Holds and replays latest value to new subscribers.

```
let subject = CurrentValueSubject<String, Never>("Initial")
subject.sink { print("First:", $0) }
subject.send("Update")
subject.sink { print("Second:", $0) } // instantly receives "Update"
```

---

3. Transforming Operators in Combine

Used to **modify values** emitted by a publisher.

`collect`

- Gathers values into an array before publishing.

```
[1,2,3,4].publisher.collect().sink { print($0) } // [1,2,3,4]
```

`map`

- Transforms each value.

```
[1,2,3].publisher.map { $0 * 2 }.sink { print($0) } // 2,4,6
```

`flatMap`

- Flattens multiple publishers into one.

```
let pub = ["A","B"].publisher
pub.flatMap { Just($0.lowercased()) }.sink { print($0) } // "a","b"
```

`scan`

- Accumulates values over time.

```
[1,2,3,4].publisher.scan(0, +).sink { print($0) }
// 1,3,6,10
```

---

4. Filtering Operators in Combine

Used to **control which values pass through**.

`filter`

```
[1,2,3,4].publisher.filter { $0 % 2 == 0 }.sink { print($0) } // 2,4
```

`removeDuplicates`

```
[1,1,2,2,3].publisher.removeDuplicates().sink { print($0) } // 1,2,3
```

`compactMap`

```
["1","a","3"].publisher.compactMap { Int($0) }.sink { print($0) } //
1,3
```

`ignoreOutput`

- Ignores all values, only completion.

`first` / `last`

- Emit only first or last value.

`dropFirst` / `drop`

```
[1,2,3,4].publisher.dropFirst(2).sink { print($0) } // 3,4
```

`prefix`

```
[1,2,3,4].publisher.prefix(2).sink { print($0) } // 1,2
```

---

5. Combining Operators in Combine

Used to **merge values from multiple publishers**.

`prepend`

```
[3,4].publisher.prepend([1,2]).sink { print($0) } // 1,2,3,4
```

append

```
[1,2].publisher.append([3,4]).sink { print($0) } // 1,2,3,4
```

switchToLatest

- Only listens to the **latest inner publisher**.

merge

- Combines values from multiple publishers.

combineLatest

```
let pub1 = PassthroughSubject<String,Never>()
let pub2 = PassthroughSubject<Int,Never>()

pub1.combineLatest(pub2).sink { print($0) }
pub1.send("A")
pub2.send(1) // ("A",1)
```

zip

- Pairs values from two publishers.

---

6. Time Manipulation Operators

debounce

- Waits until user stops emitting for a given time.
  👉 Useful for **search bars**.

`throttle`

- Emits max once every given interval.

`timeout`

- Fails if value isn't emitted within time.

---

7. Debugging Pipelines

- Use `.print()` or `.handleEvents` to log events.

```
[1,2].publisher.print("Debug").sink { print($0) }
```

---

8. share & multicast

- **share()** → Shares one subscription among multiple subscribers.
- **multicast()** → Sends values through a subject.

---

9. Future in Combine

- Represents a **one-time async operation**.

```
func asyncTask() -> Future<String, Never> {
    return Future { promise in
        DispatchQueue.global().async {
            promise(.success("Done ✅"))
        }
    }
```

```
}
asyncTask().sink { print($0) }
```

---

10. Backpressure in Combine

- Controls **rate of data flow** so subscriber isn't overwhelmed.
- Achieved using `Demand` in custom subscribers.

---

11. AnyCancellable

- Returned by `.sink` or `.assign`.
- Used to store & cancel subscriptions.

```
var cancellable: AnyCancellable?
cancellable = [1,2,3].publisher.sink { print($0) }
cancellable?.cancel()
```

🔥 **Interview Tip:**

- Always mention **Combine replaces RxSwift for Apple ecosystems**.
- Emphasize **Subjects (Passthrough / CurrentValue)**, **Operators** (map, flatMap, filter), and **AnyCancellable**.

🖥️ SwiftUI Interview Questions & Answers

---

1. What is the difference between `@State` and `@StateObject` in SwiftUI?

- **`@State`** → Used for simple local state inside a view (struct-based, lightweight).
- **`@StateObject`** → Used to create and manage the lifecycle of a reference-type (class conforming to `ObservableObject`).

```
struct CounterView: View {
    @State private var count = 0   // Local primitive state
    @StateObject private var vm = CounterViewModel() // Class state

    var body: some View {
        VStack {
            Text("Count: \(count)")
            Text("VM Count: \(vm.value)")
        }
    }
}
```

---

2. What is `@ObservedObject` and when should it be used in SwiftUI?

- Use `@ObservedObject` when you want a **view to observe a view model**, but the **view does not own its lifecycle**.
- The owner passes it in; SwiftUI will update the view when `@Published` properties change.

---

3. What is `@Published` and how does it work in SwiftUI?

- `@Published` is a **property wrapper** inside `ObservableObject` classes.

- It automatically **notifies subscribers (like SwiftUI views)** when the value changes.

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

---

4. What is `@Environment` in SwiftUI, and when should it be used?

- Provides access to **system values** (e.g., color scheme, locale, size category).
- Example:

```
@Environment(\.colorScheme) var scheme
```

---

5. What is `@EnvironmentObject` in SwiftUI, and how does it work?

- Used for **shared data across multiple views**, injected into the environment.
- Views can access it without explicitly passing down the object.

---

6. What is `@AppStorage` in SwiftUI, and how is it useful?

- A property wrapper that **persists values in UserDefaults** automatically.

```
@AppStorage("username") var username = ""
```

---

7. What is `@Binding` in SwiftUI, and when should it be used?

- Used to **create a two-way connection** between parent and child views.
- Allows a child view to modify state owned by a parent.

---

8. If a struct contains a `@Binding` property, how do you create a custom initializer?

```
struct ToggleView: View {
    @Binding var isOn: Bool

    init(isOn: Binding<Bool>) {
        self._isOn = isOn
    }
}
```

---

9. What is `@FocusedBinding` in SwiftUI, and how is it different from `@Binding`?

- `@FocusedBinding` allows binding to **focused values** (e.g., text fields).
- More dynamic compared to `@Binding`.

---

10. What is `@GestureState` in SwiftUI, and how does it work?

- Used to **track transient gesture values** during a gesture lifecycle.

```
@GestureState private var dragOffset = CGSize.zero
```

---

11. What is `@UIApplicationDelegateAdaptor` in SwiftUI, and how is it used?

- Lets you integrate an **AppDelegate** into SwiftUI apps.

```
@UIApplicationDelegateAdaptor(AppDelegate.self) var delegate
```

---

12. What are view modifiers in SwiftUI, and how are they applied?

- Functions that return a **new modified view**.
- Example: `.padding()`, `.foregroundColor()`.

---

13. What is the purpose of the `some` keyword in SwiftUI?

- Represents **opaque return type** — you don't care about the exact type, only that it conforms to `View`.

---

14. How do you create custom view modifiers in SwiftUI?

```
struct RoundedModifier: ViewModifier {
    func body(content: Content) -> some View {
        content.padding().background(Color.blue).cornerRadius(8)
    }
}
```

---

15. How do you navigate between views in SwiftUI?

- Use `NavigationStack` (iOS 16+).

```
NavigationStack {
    NavigationLink("Go", destination: DetailView())
}
```

---

16. How do you interact with UIKit components in SwiftUI?

- Use `UIViewControllerRepresentable` or `UIViewRepresentable`.

```
struct CustomView: UIViewRepresentable {
    func makeUIView(context: Context) -> UILabel { UILabel() }
    func updateUIView(_ uiView: UILabel, context: Context) {}
}
```

---

17. What is `AnyView` in SwiftUI, and when should it be used?

- Type erasure for views (use when types must match).
- Overuse can cause performance overhead.

---

18. How can you avoid using `AnyView` in SwiftUI?

- Use `@ViewBuilder` instead for type-safe conditional views.

---

19. What is `@ViewBuilder`, and how is it used?

- A result builder for constructing views conditionally or in loops.

```
@ViewBuilder
var content: some View {
    if condition { Text("True") } else { Text("False") }
}
```

🆕 New Features in Swift (Concurrency & More)

🔹 What is structured concurrency in Swift, and how does it improve async tasks?

Structured concurrency ensures that asynchronous tasks are tied to a well-defined scope.

- Child tasks are bound to their parent task (created with `async let` or `TaskGroup`).
- Ensures predictable cancellation and error propagation.
  ✅ Benefits: safer, more manageable async code compared to unstructured `DispatchQueue` or callbacks.

🔹 What are async/await, and how do they simplify asynchronous programming in Swift?

- `async` marks a function that performs asynchronous work.
- `await` suspends execution until the async task finishes.
  ✅ Benefits: turns callback-based code into linear, readable code.

```
func fetchData() async throws -> String {
    let (data, _) = try await URLSession.shared.data(from: URL(string:
"https://api.example.com")!)
    return String(data: data, encoding: .utf8) ?? ""
}
```

🔹 What is AsyncSequence, and how is it used in Swift for handling sequences of asynchronous events?

- `AsyncSequence` is like a regular `Sequence` but for async values.

- Use `for await` to iterate over values that arrive over time (e.g., network streams, notifications).

```
for await line in URL(string: "wss://chat.example.com")!.lines {
    print("Received:", line)
}
```

🔹 What are effectful read-only properties, asynchronous properties, and throwing properties in Swift?

- Swift now allows **computed properties** to be `async` and/or `throws`.
- Useful when retrieving a property involves async or fallible work.

```
struct User {
    var id: Int

    var profile: String {
        get async throws {
            try await fetchProfile(for: id)
        }
    }
}
```

🔹 What are Task and TaskGroup in Swift, and how are they used for concurrent programming?

- **`Task`**: launches a unit of work that runs concurrently.
- **`TaskGroup`**: allows spawning multiple tasks that can run in parallel and be awaited collectively.

```
await withTaskGroup(of: Int.self) { group in
    for i in 1...3 {
        group.addTask { await compute(i) }
```

```
    }
    for await result in group {
        print("Got:", result)
    }
}
```

🔹 What are Actor and GlobalActor, and how do they help manage concurrency and data safety in Swift?

- **Actor**: reference type that protects its mutable state, ensuring only one task accesses it at a time.
- Prevents data races in concurrent code.

```
actor Counter {
    private var value = 0
    func increment() { value += 1 }
    func get() -> Int { value }
}
```

- **GlobalActor**: ensures certain work always runs on the same actor (e.g., `@MainActor` ensures code runs on main thread).

🔹 What is the purpose of the @Sendable attribute, and how does it relate to the Sendable protocol in Swift?

- `@Sendable` marks closures that are safe to pass across concurrency domains.
- Ensures captured values inside the closure conform to `Sendable` (thread-safe, no data races).

```
let task = Task {
    await process { @Sendable value in
        print(value)
```

```
    }
}
```

- **Sendable protocol**: marks types that can be safely transferred across threads/actors.

👉 This is the **latest hot area in interviews (Swift 5.5 → 6)** since Apple is pushing structured concurrency and Actors heavily.

👨‍💻 Git

🔹 What Git flow have you used in past projects?

- **Git Flow**: `develop`, `feature/*`, `release/*`, `hotfix/*`, `master`.
- **GitHub Flow**: lightweight, just `main` + feature branches, merged via PR.
- **Trunk-based Development**: single `main` branch, small frequent commits.

🔹 What is the difference between merge and rebase in Git?

- **Merge**: keeps history, adds a merge commit.
- **Rebase**: replays commits on top of another branch, creates linear history.

🔹 How do you reset the last commit in Git? How can you change the last commit message?

Reset:

```
git reset --soft HEAD~1   # keep changes staged
git reset --hard HEAD~1   # remove changes completely
```

-

Change message:

```
git commit --amend -m "New commit message"
```

-

🔹 What is the cherry-pick command in Git used for?

Applies a specific commit from one branch onto another.

```
git cherry-pick <commit_hash>
```

-

🔹 What is the difference between a hard reset and a soft reset in Git?

- **Soft Reset**: moves HEAD but keeps changes staged.
- **Hard Reset**: moves HEAD and deletes changes permanently.

🔹 What is Git stash, and how is it used?

Temporarily saves uncommitted changes.

```
git stash save "WIP"
git stash pop
```

-

🏃 Agile Process

🔹 What Agile process do you follow in your projects?

- Typically **Scrum**: 2-week sprints, ceremonies (standup, planning, retro, review).

- Sometimes **Kanban**: continuous delivery with WIP limits.

🔹 How do you plan sprints in Agile?

- Define sprint goal → prioritize backlog items → assign story points → commit scope.

🔹 How do you estimate stories in Agile?

- **Story Points**: based on complexity (Fibonacci: 1,2,3,5,8).
- **Planning Poker** for consensus.

🔹 What is backlog refinement in Agile?

- Regular session to groom stories, clarify requirements, split large stories.

🔹 What is a Sprint retrospective, and why is it important?

- Meeting at sprint end → discuss **what went well, what didn't, improvements**.
- Improves team collaboration & delivery.

🔹 What is a Sprint review or showcase, and how is it conducted?

- Demo of sprint deliverables to stakeholders.
- Feedback → adjust backlog → increase transparency.

📈 Code Quality Practices

🔹 What code quality checks do you use during the PR process?

- **Static analysis**: SonarQube, SwiftLint.
- **Automated tests**: unit/UI tests.
- **Peer code reviews**.
- **CI pipelines**: ensure build passes.

🔹 What are the code review practices or guidelines followed to maintain code quality?

- Follow coding standards (Swift API Design Guidelines).
- Small PRs → easy to review.
- Test coverage required.
- No duplication, proper naming, SOLID principles.

🔐 Security Mechanisms

🔹 What are the security mechanisms used to make an app secure?

- Keychain storage, SSL pinning, ATS, secure APIs, encryption, jailbreak detection.

🔹 What is SSL pinning? What is the difference between certificate pinning and public key pinning?

- Prevents **MITM attacks** by verifying server identity.
- **Cert pinning**: pin full certificate.
- **Public key pinning**: pin public key only → more resilient to cert renewals.

🔹 How can you ensure that SSL pinning is perfectly integrated in an app?

- Test with a proxy (Charles/Fiddler) → app must reject MITM certs.

🔹 How do you securely store data in an app? What is the Keychain storage used for?

- Use **Keychain** for sensitive credentials (tokens, passwords).
- Use **UserDefaults** only for non-sensitive data.

🔹 How do you implement encryption on the database in an app?

- Use **SQLCipher** with AES encryption for CoreData/SQLite.

🔹 What are the different encryption algorithms used in app security?

- **AES** (symmetric encryption).
- **RSA** (asymmetric).
- **SHA** (hashing, integrity check).

🔹 How do you check for jailbroken devices?

- Look for suspicious paths (`/Applications/Cydia.app`).
- Check write access outside sandbox.
- Use `dyld` checks.

🔹 What is App Transport Security (ATS), and how does it contribute to app security?

- Enforces **HTTPS** connections with TLS 1.2+.
- Prevents insecure HTTP traffic.

🔹 How can you prevent screenshot capturing within an app?

On iOS, overlay secure view or use:

```
UIScreen.main.isCaptured
```

- to detect screen recording.

🔹 How can you avoid exposing confidential information, such as API keys, in a code repository?

- Use `.xcconfig` files, environment variables, or CI secrets.

🔹 What are the best practices for managing debug logs in production apps?

- Use `os_log` with levels.

- Strip sensitive logs in release builds.
- Disable verbose logging in prod.

🛠️ Managing Different Environments

🔹 **How do you manage different environments such as DEV, SIT, UAT, and Production?**

- Use **configuration files (.xcconfig)**, environment flags, and separate API endpoints.

🔹 **What is the .xcconfig file, and how is it used to manage different environments in an app?**

- External file that defines build settings.
- Example: API_BASE_URL, feature flags.

🔹 **What is the difference between a Scheme and a Target in Xcode?**

- **Target**: defines how app is built (bundle id, settings, assets).
- **Scheme**: defines build/run/debug configuration for a target.

📐 Auto Layout

❓ **What is Content Hugging Priority and Compression Resistance Priority in Auto Layout?**

- **Content Hugging Priority (CHP)**: How strongly a view resists growing *bigger* than its intrinsic size.
  👉 Higher CHP = view stays closer to its content size.

- **Compression Resistance Priority (CRP)**: How strongly a view resists being made *smaller* than its intrinsic size.
  👉 Higher CRP = view avoids being compressed.

📌 Example:

```
label.setContentHuggingPriority(.defaultHigh, for: .horizontal)

label.setContentCompressionResistancePriority(.required, for:
.horizontal)
```

This keeps the label from stretching or truncating.

🚀 Package Managers

❓ What are CocoaPods, Swift Package Manager (SPM), and Carthage? What are the differences between them?

- **CocoaPods**
    - Oldest and most widely used.
    - Uses a centralized specs repo.
    - Creates a workspace and integrates dependencies into Xcode projects.
- **Swift Package Manager (SPM)**
    - Apple's official package manager (since Swift 3+).
    - Fully integrated into Xcode.
    - No need for external tools or workspaces.

- **Carthage**
    - Decentralized dependency manager.
    - Builds frameworks you manually integrate into Xcode.
    - Lightweight, but requires manual setup.

📌 Summary:

- **SPM** → Best for modern projects.
- **CocoaPods** → Legacy projects, lots of pod support.
- **Carthage** → Lightweight/manual control.

🔔 Push Notifications

❓ What is the difference between Apple Push Notification Authentication Key (.p8) and Apple Push Notification service SSL certificate (.p12)?

- **.p12 (SSL certificate)**
    - Old mechanism.
    - Separate certs needed for Development and Production.
    - Expires and needs renewal.
- **.p8 (Auth Key)**
    - New mechanism.
    - One key works across all apps in your account.
    - No expiration. Easier to manage.

👉 Use **.p8** for modern apps (recommended by Apple).

📱 App Store Mechanism

❓ What are the different types of provisioning profiles used in iOS development?

1. **Development** – For local builds + debugging on registered devices.
2. **Ad Hoc** – For distributing test builds outside the App Store, limited to registered devices.
3. **Enterprise** – For in-house distribution within a company.
4. **App Store** – For publishing apps on the App Store.

❓ What are the different certificate types used in iOS development?

- **Development Certificate** → For code signing apps in dev environment.
- **Distribution Certificate** → For signing apps for App Store / Ad Hoc / Enterprise.

❓ How does TestFlight beta testing work, and what is the difference between internal and external testing?

- **Internal Testing**
  - Up to 100 team members.
  - Immediate access after upload.
- **External Testing**
  - Up to 10,000 testers.
  - Requires App Review approval before testers can use.

🧪 Unit Testing with XCTest

❓ How do you use the XCTest framework for Unit Testing in Swift?

```
import XCTest

@testable import MyApp


class MathTests: XCTestCase {

    func testAddition() {

        XCTAssertEqual(2 + 3, 5)

    }

}
```

Run tests in **Product** → **Test** or `Cmd+U`.

❓ How do you manage real and mock implementations in API testing?

- Use **protocol-based abstraction** + **Dependency Injection**.

```
protocol APIService {

    func fetchData() -> String

}


class RealAPIService: APIService { ... }
```

```
class MockAPIService: APIService { ... }
```

Inject `MockAPIService` during tests.

❓ What are the different types of assertions used in Unit Testing?

- `XCTAssertEqual`, `XCTAssertTrue`, `XCTAssertFalse`, `XCTAssertNil`, `XCTAssertNotNil`, `XCTAssertThrowsError`.

❓ What is XCTSkipUnless used for in Unit Testing?

Skips a test unless a condition is true.

```
XCTSkipUnless(userIsLoggedIn, "User must be logged in to test profile feature")
```

❓ How does Dependency Injection help with Unit Testing?

- Decouples dependencies → makes testing easier.
- Swap real implementations with mocks/fakes.

❓ How do you test asynchronous tasks or real API integration in Swift?

Use `expectation` and `wait`.

```
func testAsyncAPI() {

    let exp = expectation(description: "API completes")

    fetchData { result in
```

```
        XCTAssertNotNil(result)
        exp.fulfill()
    }
    wait(for: [exp], timeout: 2.0)
}
```

❓ What is the use case of tearDown() and setup() methods in testing?

- `setUp()` → Runs **before each test**. Initialize test data.
- `tearDown()` → Runs **after each test**. Clean up resources.

❓ What is behavior-driven testing, and how do you write behavior-driven tests in iOS?

- Describes app behavior in **human-readable format**.
- Uses **BDD frameworks like Quick + Nimble**.

```
describe("Calculator") {
    it("adds numbers correctly") {
        expect(2 + 3).to(equal(5))
    }
}
```

❓ What is the Quick and Nimble framework, and how is it used for Unit Testing?

- **Quick** → BDD-style syntax.
- **Nimble** → Matchers like `to(equal())`, `to(beNil())`.

❓ Is setUpWithError() called before each test or only before the first test?

👉 **Before each test**.

❓ Is tearDownWithError() called after each test?

👉 **Yes, after each test**.

❓ What is SnapshotTesting, and which framework is used for it?

- Testing UI by comparing rendered output with a saved snapshot.
- Framework: pointfreeco/swift-snapshot-testing.

```
assertSnapshot(matching: myView, as: .image)
```

❓ What is code coverage in testing?

- % of app code executed by tests.
- Xcode → **Test Navigator** → **Show Code Coverage**.
- Goal: High coverage, but prioritize **quality tests over %**.

🧱 SOLID Principles in Swift

1. Single Responsibility Principle (SRP)

👉 A class should have only **one reason to change**, i.e., one responsibility.

```
// ❌ Bad: User handles login + saving to DB
class UserManager {
    func login(username: String, password: String) {}
    func saveToDatabase(user: String) {}
}

// ✅ Good: Split into responsibilities
class AuthService {
    func login(username: String, password: String) {}
}

class UserRepository {
    func save(user: String) {}
}
```

---

2. Open/Closed Principle (OCP)

👉 Classes should be **open for extension but closed for modification**.

```
// ❌ Bad: Modifying class for new discount
class Discount {
    func calculate(price: Double, type: String) -> Double {
```

```
        if type == "Student" { return price * 0.9 }
        if type == "Senior" { return price * 0.8 }
        return price
    }
}

// ✅ Good: Extend using polymorphism
protocol DiscountStrategy {
    func apply(price: Double) -> Double
}

class StudentDiscount: DiscountStrategy {
    func apply(price: Double) -> Double { price * 0.9 }
}

class SeniorDiscount: DiscountStrategy {
    func apply(price: Double) -> Double { price * 0.8 }
}
```

---

3. Liskov Substitution Principle (LSP)

👉 Subtypes must be substitutable for their base types.

```
class Bird {
```

```
    func fly() {}
}

// ❌ Bad: Penguin violates LSP because it cannot fly
class Penguin: Bird {
    override func fly() { fatalError("Penguins can't fly") }
}

// ✅ Good: Separate behaviors
protocol Flyable { func fly() }

class Sparrow: Flyable {
    func fly() { print("Flying high") }
}

class Penguin {
    func swim() { print("Swimming") }
}
```

---

4. Interface Segregation Principle (ISP)

👉 Clients should not be forced to depend on interfaces they don't use.

```
// ❌ Bad: One big protocol
```

```
protocol Worker {
    func code()
    func test()
    func design()
}

// ✅ Good: Smaller protocols
protocol Coder { func code() }
protocol Tester { func test() }
protocol Designer { func design() }

class Developer: Coder { func code() {} }
class QA: Tester { func test() {} }
```

---

5. Dependency Inversion Principle (DIP)

👉 Depend on **abstractions**, not on concrete implementations.

```
// ❌ Bad: Tightly coupled
class MySQLDatabase {
    func save(data: String) {}
}

class UserService {
```

```
    let db = MySQLDatabase()
    func saveUser(name: String) { db.save(data: name) }
}

// ✅ Good: Use abstraction
protocol Database {
    func save(data: String)
}

class MySQLDatabase: Database {
    func save(data: String) {}
}

class UserService {
    let db: Database
    init(db: Database) { self.db = db }
    func saveUser(name: String) { db.save(data: name) }
}
```

💉 Dependency Injection in Swift

1. What is Dependency Injection?

It's a **design pattern** where dependencies (services, objects) are provided to a class rather than being created inside it. This improves **testability and flexibility**.

---

2. Types of Dependency Injection

1. **Constructor Injection** (via `init`)

```
class Service {
    func run() { print("Running service") }
}

class App {
    let service: Service
    init(service: Service) { self.service = service }
}

let app = App(service: Service())
```

2. **Property Injection**

```
class App {
    var service: Service?
}
let app = App()
app.service = Service()
```

3. **Method Injection**

```
class App {
    func start(service: Service) {
        service.run()
    }
}
```

---

3. Why use Dependency Injection?

- Reduces **tight coupling**.
- Improves **testability** (e.g., mock services in unit tests).
- Improves **code maintainability**.

---

4. Resolver

A **Swift DI framework** that provides automatic dependency resolution.

```
import Resolver


class Service {
    func run() { print("Running") }
}
```

```
class App {
    @Injected var service: Service
}
```

---

5. SwiftInject

Another DI framework similar to Resolver, often used for **complex dependency graphs**.

```
import SwiftInject

class Service {}
class App {
    let service: Service
    init(service: Service) { self.service = service }
}
```

🧬 Core Swift Concepts

1. Composition over Inheritance

👉 Prefer combining behaviors via **protocols/structs** rather than deep inheritance.

```
// ❌ Inheritance
class Dog {
    func bark() {}
```

```
}

class GuardDog: Dog {

    func guardHouse() {}

}


// ✅ Composition

protocol Barking { func bark() }

protocol Guarding { func guardHouse() }


class Dog: Barking {

    func bark() {}

}


class GuardDog: Barking, Guarding {

    func bark() {}

    func guardHouse() {}

}
```

---

2. Pure Function

👉 A function is **pure** if:

- Output depends **only** on input.

- No **side effects** (e.g., no global state change).

```
// ✅ Pure Function
func add(a: Int, b: Int) -> Int {
    return a + b
}

// ❌ Impure (changes global state)
var counter = 0
func increment() -> Int {
    counter += 1
    return counter
}
```

---

1. 🚀 Major Updates in iOS 26

(as of 2025, based on Apple's WWDC announcements)

- **AI Integration (Apple Intelligence)** – Deep system-wide integration of on-device AI for text summarization, smart replies, and image generation.
- **Siri Overhaul** – Context-aware Siri with conversational memory and tighter integration across apps.
- **Customizable Control Center** – Multi-page layouts, resizable toggles, and per-app quick actions.

- **Enhanced Privacy Features** – More granular permissions (e.g., temporary location sharing, app tracker transparency upgrades).
- **Health & Fitness** – Expanded health tracking, fertility cycle predictions, and mental health insights.
- **Mail & Messages** – Smart categorization (Promotions, Transactions, Personal), improved scheduling, and AI-based reply suggestions.
- **Safari** – AI-powered content highlights and improved Reader mode.
- **Developers** – Swift 6 support, new Swift macros, and advanced ARKit + RealityKit APIs for Vision Pro integration.

---

2. 🆚 Swift 5 vs Swift 6

Feature	Swift 5	Swift 6
Concurrency	Introduced structured concurrency (async/await, Task, Actor)	Enhanced with stricter `Sendable` checking, GlobalActors improvements
Macros	Not available	Introduced Swift Macros for boilerplate reduction
Memory Safety	Stable, but less strict for concurrency	Enforced stricter concurrency safety rules
Backward Compatibility	Strong backward compatibility with Swift 4	Some breaking changes due to stricter concurrency rules

Performance	Optimized, but manual in some cases	Better compiler optimizations with macros + concurrency
Error Handling	Sync & async throwing supported	Expanded to support async sequences and effectful properties

Example (Swift 6 Macro):

```
@freestanding(expression)
macro uppercased(_ value: String) -> String = {
    return value.uppercased()
}

let name = #uppercased("dk")  // Output: "DK"
```

---

3. 🏗️ Class vs Actor vs Struct vs Enum in Swift

Aspect	Class	Actor	Struct	Enum
Type	Reference	Reference (concurrent-safe)	Value	Value
Inheritance	✅ Supported	❌ (No subclassing)	❌	❌

Mutability	Mutable	Mutable but thread-safe	Immutable by default	Immutable by default
Thread Safety	❌ Needs manual sync	✅ Built-in thread safety	✅ Safe (value semantics)	✅ Safe
Use Case	Complex models, shared objects	Concurrency-safe shared objects	Lightweight models, performance-critical	Represent fixed states/cases

Examples:

👉 **Class**

```
class Car {
    var model: String
    init(model: String) { self.model = model }
}
```

👉 **Actor**

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

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

👉 **Struct**

```
struct Point {
    var x: Int
    var y: Int
}
```

👉 **Enum**

```
enum Direction {
    case north, south, east, west
}
```

---

4. List of WWDC 2025 Updates

Here's a curated overview of the major announcements from **Apple's WWDC 2025**, complete with noteworthy details:

- **Unified Liquid Glass Design Across All Platforms**
  Apple unveiled **Liquid Glass**, a dynamic, translucent design language featured in **iOS 26, iPadOS 26, macOS Tahoe, watchOS 26, tvOS 26, and visionOS 26** — delivering a cohesive visual style across devices.

- **Year-Based OS Naming Convention Introduced**
  Apple transitioned away from numbered OS versions to a **year-based naming system**, including **iOS 26, iPadOS 26, macOS 26, watchOS 26, tvOS 26,** and **visionOS 26**.

- **Enhanced App Functionalities & AI Integration**

    - Revamped **Camera, Safari, and Phone** apps with updated layouts and combined Recents/Favorites/Voicemail views.

    - **Messages** gains custom backgrounds, group polling, typing indicators, and spam filtering.

    - **Live Translation powered by Apple Intelligence** now available in **Messages, FaceTime, and Phone** — all local on-device for privacy.

- **Developer Opening: On-Device AI via Foundation Models API**
  Apple offers developers access to **on-device large language models** via the **Foundation Models framework**, enabling features like screenshot-based visual intelligence and AI-powered shortcuts integration.

- **iPadOS Multitasking Upgrades & New Journal App**
  iPadOS 26 introduces advanced windowing—resizable, movable app windows and a macOS-like Preview app.
  Plus, the **Journal** app makes its debut on iPad and Mac, offering rich-media journaling with cross-device syncing.

- **macOS Tahoe Enhancements**
  Enhanced **Spotlight** with action integration (like sending emails), and expanded OS functionality on the Mac, including access to Phone and Games apps.

- **New Games App & VisionOS Controller Support**
  Apple launched a dedicated **Games app** to centralize access to your games library and social features.
  visionOS 26 adds support for **PSVR2 controllers** and spatial widgets — improving immersion and usability.

- **watchOS Gesture & Workout AI Upgrades**
  watchOS 26 gets a “wrist-flick” gesture for notifications and an **AI-powered Workout Buddy** that offers dynamic fitness guidance.

- **AirPods Get Smart Camera Control & Voice Isolation**
  AirPods now support **camera control via tap** on the stem and improved voice isolation for high-quality vocal recording in noisy settings.

Associated Type in Swift

🔹 Definition

- An **associated type** is a placeholder type used inside a **protocol**.
- It allows protocols to be **generic**, without specifying an exact type until a type conforms to the protocol.
- Think of it as saying: *"Any type can be used here, but I'll tell you later when a concrete type conforms."*

🔹 Syntax

```
protocol Container {

    associatedtype Item  // placeholder type

    func add(_ item: Item)

    func getAll() -> [Item]

}
```

🔹 Example

```
// Protocol with associated type

protocol Container {

    associatedtype Item

    func add(_ item: Item)

    func getAll() -> [Item]

}
```

```
// Conforming with Int
class IntContainer: Container {
    private var items: [Int] = []

    func add(_ item: Int) {
        items.append(item)
    }

    func getAll() -> [Int] {
        return items
    }
}

// Conforming with String
class StringContainer: Container {
    private var items: [String] = []

    func add(_ item: String) {
        items.append(item)
    }

    func getAll() -> [String] {
```

```
        return items
    }
}

let intBox = IntContainer()
intBox.add(10)
intBox.add(20)
print(intBox.getAll())  // [10, 20]

let strBox = StringContainer()
strBox.add("Hello")
strBox.add("Swift")
print(strBox.getAll())  // ["Hello", "Swift"]
```

🔹 When to Use

- When designing **protocols that work with any type**, but don't know the type upfront.
- For **collections**, **iterators**, or **generic abstractions**.

🔹 Difference Between `associatedtype` vs. Generics

- **Generics (<T>)**: Declared at **function/class/struct/enum** level.
- **Associated Type**: Declared inside a **protocol**, letting conforming types decide later.

```
// Generic Example
func printItems<T>(items: [T]) {
    for item in items {
        print(item)
    }
}

// Associated Type Example
protocol Iterator {
    associatedtype Element
    mutating func next() -> Element?
}
```

🔹 What is an Opaque Type?

- An **opaque type** hides the concrete return type from the caller but guarantees a specific protocol it conforms to.
- Declared using the keyword `some`.
- The function defines the exact type internally, but the caller only knows “this type conforms to X protocol”.

🔹 Syntax

```
func makeShape() -> some Shape {
    return Circle()
}
```

Here:

- The caller knows the return type conforms to `Shape`.
- The exact type (`Circle`) is hidden.

🔹 Example 1: Basic Swift Opaque Type

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

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

struct Square: Shape {
    var side: Double
    func area() -> Double { side * side }
}
```

```
// Opaque type return
func createShape() -> some Shape {
    return Circle(radius: 5)  // concrete type is hidden
}

let s = createShape()
print(s.area()) // ✅ works
```

If we used `-> Shape` instead of `-> some Shape`, it would be a **protocol type** (existential), which allows multiple conforming types but loses compile-time optimizations.
