# iOS Interview Prep Pack

## About this pack

This interview pack is designed for junior and mid–level iOS developers preparing for technical interviews. It contains more than 100 curated questions with concise answers covering protocols, memory management (ARC), multithreading, Combine, and SwiftUI. It also includes 10 coding tasks with sample solutions and a cheat sheet for key architectural concepts like SOLID, MVVM and VIPER. Use this as a focused study guide to brush up on core concepts and practice coding problems in Swift.

---

## Q&A Section

### Protocols and Generics (Q1–Q20)

1. **What is a protocol?** A protocol defines a blueprint of methods, properties and other requirements for tasks or pieces of functionality. Types conforming to a protocol provide concrete implementations of those requirements.
2. **When should you use an associated type in a protocol?** Use associatedtype when a protocol needs to refer to a placeholder type that will be specified by the conforming type, such as in a generic container or iterator.
3. **How do you constrain a protocol to classes only?** Add : AnyObject to the protocol declaration (e.g. protocol MyDelegate: AnyObject { … }). This allows the protocol to be adopted only by class types and lets you declare weak references to delegates.
4. **What is protocol inheritance?** A protocol can inherit from one or more protocols to reuse their requirements. For example, protocol Animatable: Drawable { … } inherits all requirements from Drawable.
5. **How can you provide default implementations for protocol methods?** You can write an extension on the protocol and implement the method there. Conforming types receive the default implementation automatically unless they override it.
6. **What’s the difference between protocol composition and inheritance?** Protocol composition (typealias P = A & B) is a temporary requirement that a type must conform to multiple protocols at once, whereas inheritance permanently builds one protocol on top of another.
7. **When would you choose composition over inheritance?** Composition is preferred when behaviour can be built from small, reusable pieces; inheritance is appropriate when modelling an is-a relationship and reusing implementation.
8. **What is type erasure?** Type erasure hides the concrete type behind a wrapper that conforms to a protocol. It solves the problem of returning protocol types with associated types, such as AnySequence or AnyPublisher.

9. **What does Equatable provide?** Conforming to Equatable allows instances to be compared using == and !=. The compiler can synthesize conformance for many structures if all stored properties are Equatable.
10. **How does Comparable build on Equatable?** Comparable adds <, <=, >, and >= methods for ordering. The compiler can often synthesize it from a single implementation of <.
11. **Why conform to Hashable?** Hashable is required when using a type as a key in a dictionary or storing it in a set. It produces a hash value consistent with equality.
12. **What is Identifiable used for?** In SwiftUI, conforming to Identifiable provides a stable id property used by views like List to differentiate items.
13. **When should you use CaseIterable?** For enums without associated values, CaseIterable synthesizes an allCases array, which is useful for iterating over all cases.
14. **How do you provide a custom string representation?** Conform to CustomStringConvertible and implement the description property.
15. **What's the difference between Codable and Encodable?** Codable is a type alias for Encodable & Decodable. Conform to Encodable when you only need to encode, and Decodable when you only need to decode.
16. **Provide an example of a delegate protocol.** A delegate protocol defines methods that a delegate object must implement. For instance:

```
protocol DownloadDelegate: AnyObject {
  func downloadDidFinish(_ data: Data)
}
class Downloader {
  weak var delegate: DownloadDelegate?
  func start() {
    // after downloading
    delegate?.downloadDidFinish(Data())
  }
}
```

17. **How do you restrict protocol adoption to classes only?** Use protocol P: AnyObject { } so that only class types can conform; this allows the use of weak references.
18. **What does Self requirement mean in protocols?** Using Self in a protocol means the conforming type must return itself; it makes the protocol non-existential and cannot be used as a type (e.g., protocol Copyable { func copy() -> Self }).
19. **Show an example of protocol-oriented dependency injection.** Define a protocol for the dependency and inject it via initializer:

```
protocol NetworkService { func requestData() async throws -> Data }
class APIClient: NetworkService { /* ... */ }
class ViewModel {
  private let service: NetworkService
```

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

20. **Why prefer composition over inheritance with protocols?** Protocols enable multiple conformances, decoupling and testing. They avoid the tight coupling and fragility often associated with deep class hierarchies.

## Memory Management (ARC) (Q21–Q40)

21. **What is ARC?** Automatic Reference Counting is Swift's memory management system. It automatically increments and decrements the retain count of class instances and deallocates them when there are no strong references.
22. **Explain the difference between strong, weak, and unowned.** Strong references keep an object alive. Weak references do not keep an object alive and become nil when the object is deallocated. Unowned references do not keep an object alive but assume the referenced object will never be nil (unsafe if deallocated first).
23. **When would you choose unowned over weak?** Use unowned when the referenced object has the same or longer lifetime than the owner and should never be nil, such as a CreditCard holding an unowned reference to its Customer.
24. **Describe a strong reference cycle.** A cycle occurs when two objects hold strong references to each other, preventing ARC from deallocating them. Example: a Person with a strong Apartment property and an Apartment with a strong Person tenant.
25. **How do you break a reference cycle between two classes?** Declare one of the references as weak or unowned. For example, make the Apartment's tenant property weak.
26. **What is a capture list?** A capture list defines how captured variables are used in a closure. Use [weak self] or [unowned self] to avoid strong reference cycles with closures.
27. **What is an autorelease pool?** In Objective-C, an autorelease pool collects autoreleased objects and releases them when the pool is drained. Use @autoreleasepool in loops to reduce peak memory usage.
28. **How can you debug memory leaks?** Use Xcode's Instruments (Leaks and Allocations) to detect leaks. Check for strong reference cycles in classes and closures.
29. **Are structs managed by ARC?** No. Structs are value types and are stored on the stack when possible. They are copied by value and do not require retain counts.
30. **Can closures cause memory leaks?** Yes. If a closure captures self strongly and both hold references to each other, a retain cycle occurs. Use capture lists to break the cycle.
31. **What is the ARC side table?** The side table is an internal data structure holding metadata like reference counts, weak/unowned reference lists and associated objects. It's created on demand for each object.
32. **When does ARC deallocate an object?** An object is deallocated when its retain count drops to zero—that is, when there are no strong references to it.
33. **How do you avoid a strong reference cycle in closures?** Use [weak self] in the capture list for closures stored or executed later (escaping closures). Alternatively use [unowned self] when self will outlive the closure.

34. **What is the difference between strong and unowned reference cycles?** A strong cycle keeps both objects alive, while an unowned cycle crashes at runtime when the object has been deallocated and the unowned reference is accessed.
35. **When can you use an optional unowned reference?** Rarely. Declare it as unowned var delegate: MyDelegate?. It behaves like weak, but you assert that the delegate will not be set to nil until after you clear it.
36. **Why declare delegates as weak?** Delegates are usually held by their owner (e.g., view controllers by views). Declaring the delegate weak prevents a cycle where the delegate strongly retains the owner.
37. **How do you implement a deinitializer?** In classes, implement deinit { … } to perform cleanup. This method is called automatically just before the instance is deallocated.
38. **How is deinit different from C++ destructors?** deinit cannot be called manually and is guaranteed to be called once. You cannot throw errors or control when it runs beyond removing all strong references.
39. **Give an example of manual memory management in Objective-C.** In manual retain/release mode, you allocate objects using alloc/init, call retain to take ownership, and call release or autorelease when finished.
40. **What happens if you over-release an object?** Over-releasing leads to a crash because ARC attempts to deallocate an already deallocated object. Conversely, failing to release causes memory leaks.

## Concurrency and Multithreading (Q41–Q60)

41. **What is the difference between concurrency and parallelism?** Concurrency is about managing multiple tasks at once by interleaving their execution. Parallelism is about running multiple tasks at the same time on different cores.
42. **How do threads and processes differ?** A process is an independent program with its own memory space. A thread is a lightweight unit of execution within a process, sharing memory with other threads.
43. **What is GCD?** Grand Central Dispatch is a low-level C API for managing concurrent code via dispatch queues. It handles thread creation and scheduling for you.
44. **How do serial and concurrent queues differ?** A serial queue executes tasks one at a time in order. A concurrent queue starts tasks in order but allows multiple tasks to run simultaneously.
45. **What is a race condition?** A bug where the result depends on the timing of multiple threads accessing shared data without proper synchronization.
46. **Define deadlock.** A situation where two or more tasks are blocked forever waiting for each other to release resources.
47. **Define livelock.** Tasks are not blocked but keep repeating work and yielding, making no progress because they interfere with each other.
48. **What is starvation?** When a task never gets access to the resource it needs because other tasks monopolize it.
49. **Explain priority inversion.** A low-priority task holds a resource needed by a high-priority task while a medium-priority task preempts the low-priority task, effectively blocking the high-priority task.

50. **What are actors in Swift concurrency?** Actors are reference types that protect their mutable state. Only one actor method can access its state at a time, preventing data races.
51. **What is structured concurrency?** A model where tasks are arranged in a hierarchy: parent tasks manage child tasks and wait for them to complete. async let and Task use this model.
52. **When would you use async let?** Use async let to run multiple asynchronous expressions concurrently and wait for their results later.
53. **What is a dispatch group?** A GCD mechanism that allows you to aggregate a collection of tasks and be notified when they all complete.
54. **How does a barrier work in GCD?** A barrier flag ensures that a block is executed exclusively on a concurrent queue, allowing safe mutation of shared resources.
55. **When should you dispatch work to the main queue?** UI updates must happen on the main thread. Use DispatchQueue.main.async to schedule UI work.
56. **What is Task.detached?** It creates a detached task that is independent of the current actor or task context. Use it for tasks that should not inherit context.
57. **Why avoid long-running operations on the main thread?** Doing heavy work on the main thread freezes the UI and leads to a poor user experience.
58. **What's the difference between synchronous and asynchronous execution?** Synchronous execution blocks the current thread until completion, whereas asynchronous execution returns immediately and executes later.
59. **How do you cancel a Task?** Keep a reference to the Task and call its cancel() method. Within the task, periodically check Task.isCancelled and handle cancellation.
60. **What does @Sendable ensure?** It indicates that a closure can be executed concurrently. The compiler checks that captured variables are safe for concurrent use.

## Combine (Q61–Q80)

61. **What is Combine?** Combine is Apple's reactive framework that provides a declarative Swift API for processing values over time.
62. **Define publisher and subscriber.** A publisher produces a sequence of values over time. A subscriber receives those values and handles completion or failure.
63. **PassthroughSubject vs CurrentValueSubject?** PassthroughSubject broadcasts new values to subscribers and does not store the latest value. CurrentValueSubject stores the current value and emits it immediately upon subscription.
64. **What is an operator?** Operators transform, filter or combine publisher output. Examples include map, filter, debounce and combineLatest.
65. **How does map work?** It transforms each output value from a publisher into another value using a closure.
66. **What does filter do?** It only forwards values that satisfy a given predicate.
67. **When to use debounce?** To wait for a pause in values before forwarding the latest value, typically used with user input to reduce network requests.
68. **What is combineLatest?** It combines the latest values of two publishers and emits a tuple whenever either publishes a value.

69. **Difference between sink and assign.** sink lets you handle values and completions with closures. assign writes values directly to a property on an object.
70. **How do you handle errors in Combine?** Use catch to recover from errors or transform them into values. Publishers can also fail with an error type.
71. **What does flatMap do?** It transforms each value into a new publisher and flattens the resulting stream.
72. **How do you cancel a Combine subscription?** Store the AnyCancellable returned by sink or assign and call cancel() when needed.
73. **What is @Published used for?** A property wrapper that exposes a publisher whenever the property changes. Often used in SwiftUI view models.
74. **Show an example of chaining operators.** For instance: publisher.map { $0 * 2 }.filter { $0 > 10 }.sink(...).
75. **What's the difference between share and multicast?** share turns a publisher into a shared publisher that multicasts to multiple subscribers. multicast uses a user-provided subject to share subscription and supports manual connection.
76. **What's the difference between a subject and a publisher?** Subjects are both publishers and subscribers; they can send values manually. A publisher is read-only.
77. **Why use eraseToAnyPublisher?** It hides the concrete type of a publisher and returns AnyPublisher, making it easier to expose publishers without revealing implementation details.
78. **How do you extend a publisher?** You can add custom operators by writing an extension on Publisher and returning a new publisher using map or flatMap.
79. **What role do schedulers play in Combine?** Schedulers determine the thread or queue on which a publisher delivers values. Use receive(on:) and subscribe(on:) to control execution context.
80. **Give an example of using Combine to fetch network data.** You can wrap a URLSession data task in a publisher: URLSession.shared.dataTaskPublisher(for: url).map(\ .data).decode(type: Model.self, decoder: JSONDecoder()).eraseToAnyPublisher().

## SwiftUI (Q81–Q100)

81. **Why are SwiftUI views structs instead of classes?** Structs are value types; they have predictable behaviour, are inexpensive to create and encourage a declarative, immutable view tree.
82. **What's the difference between @State and @Binding?** @State declares a source of truth owned by the view. @Binding is a reference to a state owned by a parent view, allowing two-way binding.
83. **@ObservedObject vs @StateObject vs @EnvironmentObject?** Use @StateObject to create and own an observable object. Use @ObservedObject to observe an existing object provided from outside. Use @EnvironmentObject to access shared data injected into the environment.
84. **When are view lifecycle callbacks called?** SwiftUI provides onAppear and onDisappear closures that run when a view appears or disappears in the UI hierarchy.

85. **What is @AppStorage?** A property wrapper that reads and writes to UserDefaults and automatically updates the view when the value changes.
86. **How do you create a list of items?** Use List with a collection and an id key path or Identifiable conformance. For example: List(items) { item in Text(item.name) }.
87. **How do you perform navigation in SwiftUI?** Use NavigationLink inside a NavigationView to push a destination view onto the navigation stack.
88. **What's the difference between VStack, HStack and ZStack?** They arrange child views vertically, horizontally and by overlaying them respectively.
89. **Why is the order of modifiers important?** Modifiers return new views. The order affects which view each modifier applies to; for example, .padding().background(Color.red) differs from .background(Color.red).padding().
90. **onAppear vs .task?** onAppear executes a synchronous closure when a view appears. .task launches an asynchronous task when the view appears and supports async/await.
91. **What is @Environment used for?** It reads values from the environment, such as colour scheme, locale or managed object context.
92. **How do you animate state changes?** Wrap state changes in withAnimation { … } or use implicit animations via .animation(_:) on a view.
93. **What is GeometryReader?** A container that gives access to the geometry proxy, enabling layout calculations based on size and coordinate space.
94. **ForEach vs List?** ForEach is a generic structure that repeats views. List wraps a scrollable list with built-in row separators, editing and indexing features.
95. **How do you integrate UIKit with SwiftUI?** Use UIViewRepresentable or UIViewControllerRepresentable to wrap UIKit components inside SwiftUI views.
96. **What's a PreferenceKey?** A mechanism to propagate values up the view hierarchy, used for reading child view values from an ancestor.
97. **sheet vs fullScreenCover?** Both present modal views. sheet sizes its content to a fraction of the screen on iPad and macOS; fullScreenCover always covers the full screen.
98. **How does Combine interact with SwiftUI?** Use @Published properties in an ObservableObject and mark the view with @StateObject or @ObservedObject. SwiftUI automatically updates when publishers emit new values.
99. **How do you unit test SwiftUI views?** Use XCTAssert with UIViewController hosting the view or XCTAssert with snapshotTesting frameworks. Check if the view hierarchy contains expected subviews and states.
100. **What is ScenePhase detection?** It provides the current lifecycle phase of your app (active, background, inactive). Use @Environment(\.scenePhase) to respond to changes such as entering the background.

---

## Coding Tasks

Below are 10 coding tasks with brief descriptions and links to sample solutions included in this pack. Each solution is provided as a standalone Swift file.

| Task | Description | Solution File |
|---|---|---|
| 1. Generic Stack | Implement a generic stack with push, pop, and peek methods. | stack.swift |
| 2. Reverse Linked List | Reverse a singly linked list and return the new head. | linkedList.swift |
| 3. Find Duplicates | Write a function that returns all duplicate integers in an array. | duplicates.swift |
| 4. LRU Cache | Implement an LRU cache with get and put operations. | lru_cache.swift |
| 5. Combine Example | Build a pipeline that doubles numbers from a publisher and prints them. | combine_example.swift |
| 6. SwiftUI Counter | Create a simple counter app using SwiftUI with a button to increment a count. | swiftui_counter.swift |
| 7. Delegate Pattern | Demonstrate a delegate pattern using a weak delegate to avoid memory leaks. | delegate_memory.swift |
| 8. JSON Parsing | Decode a JSON string into a Swift model using Codable. | json_parsing.swift |
| 9. Async Fetch | Concurrently fetch data from two URLs using async/await and combine results. | async_fetch.swift |
| 10. Parallel Sum | Use GCD to sum an array of numbers in parallel using DispatchGroup. | parallel_sum.swift |

## Cheat Sheet

### SOLID Principles

- **Single Responsibility** – Each class or module should have one reason to change.
- **Open/Closed** – Software entities should be open for extension but closed for modification.
- **Liskov Substitution** – Objects of a superclass should be replaceable with objects of a subclass without affecting correctness.
- **Interface Segregation** – Prefer many client-specific interfaces to one general purpose interface.
- **Dependency Inversion** – Depend on abstractions, not on concrete implementations.

### MVVM

- **Model** – Represents the app’s data and business logic. It should be framework-agnostic.
- **View** – Declares the UI and reflects the state of the view model. In SwiftUI this is a struct conforming to View.
- **ViewModel** – Transforms model data into values the view can display. It exposes bindable properties using @Published and receives user intents via methods.
- MVVM improves testability and separation of concerns compared to MVC by moving presentation logic out of view controllers.

### VIPER

- **View** – Displays data and handles user interaction.
- **Interactor** – Contains business logic and interacts with data managers or API services.
- **Presenter** – Transforms data from the interactor to view models and triggers navigation via the router.
- **Entity** – Plain data objects representing business models.
- **Router** – Handles navigation and module assembly. VIPER emphasizes separation of concerns but can be more verbose than MVVM.

---

## How to Use

Use the Q&A section as a flash-card style review. Read each question and recall the answer, then verify your response. Work through the coding tasks in Xcode or a Playground to practice problem solving and sharpen your Swift syntax. Refer to the cheat sheet to refresh foundational architectural principles during interviews.