iOS Junior Level Interview Questions

> *iOS Interview Questions - Your Cheat Sheet For iOS Interview*

Prepared by Shobhakar Tiwari

IOS DEVELOPER ADVOCATE | MOCK INTERVIEW COACH
Developing for Apple platforms
Guiding iOS developers to interview success
DM FOR iOS MOCK Session

Swift Interview Questions

Welcome to the Swift Interview Questions repository! This document contains a curated list of essential interview questions that can help you prepare for iOS developer roles focusing on Swift programming. Each question is designed to test your knowledge of the Swift language, its features, and best practices.

Questions

1. What is Swift language? Is it an OOP language or POP (Protocol Oriented Programming)?

Answer: Swift is a multi-paradigm programming language that supports both Object-Oriented Programming (OOP) and Protocol-Oriented Programming (POP).

2. Why is Swift called Protocol Oriented Programming?

Answer: Swift emphasizes the use of protocols to define blueprints of methods, properties, and other requirements, promoting a flexible and modular design.

3. Why is Swift called a strictly type-safe language?

Answer: Swift enforces strong type-checking at compile time, reducing runtime crashes and errors by ensuring that variables are used with the correct data types.

4. What is Optional?

Answer: An Optional in Swift is a type that can hold either a value or `nil`, indicating the absence of a value.

5. Is Optional an ENUM or STRUCT?

Answer: Optional is an enumeration (ENUM) that has two cases: `.none` (for `nil`) and `.some(Wrapped)`.

6. What are the different ways of getting data from optionals?

Answer: You can retrieve data from optionals using optional binding (`if let`, `guard let`), forced unwrapping (`!`), or optional chaining.

7. What are the differences between `guard let` vs `if let`? When would you use `guard let` over `if let`?

Answer: `guard let` is used for early exits from a function or loop if the condition is not met, while `if let` allows you to handle the case where an optional contains a value. Use `guard let` for cleaner and more readable code when you need to exit early.

8. What is the force unwrapping operator and why is it not recommended?

Answer: The force unwrapping operator (`!`) is used to retrieve the value of an optional. It is not recommended because if the optional is `nil`, it will cause a runtime crash.

9. What is a mutable structure?

Answer: A mutable structure in Swift is a structure whose properties can be modified after its initial creation.

10. What is the `mutating` keyword and why is it needed?

Answer: The `mutating` keyword is used in methods of structures to indicate that the method can modify the properties of the structure itself.

11. What is a closure? Is a closure also a function?

Answer: A closure is a self-contained block of functionality that can be passed around and used in your code. Yes, closures can be thought of as anonymous functions.

12. Is closure a reference type or value type?

Answer: A closure is a reference type, which means it captures variables and constants from its surrounding context.

13. Create a function for finding the sum of two numbers and convert this into a sum closure.

```swift
func sum(a: Int, b: Int) -> Int {
    return a + b
}

let sumClosure: (Int, Int) -> Int = { a, b in
    return a + b
}
```

14. Create a structure of an Employee and within this, there should be one Department where the Department has an id and name.

Answer:

```swift
struct Department {
    var id: Int
    var name: String
}

struct Employee {
    var id: Int
    var name: String
    var department: Department
}
```

15. What are access modifiers?

Answer: Access modifiers define the visibility of properties and methods in Swift, including public, private, fileprivate, and internal.

16. What is the difference between SceneDelegate and AppDelegate?

Answer: AppDelegate manages app-wide behavior, while SceneDelegate manages individual instances of UI. SceneDelegate allows multiple UI scenes to run simultaneously.

17. What is the use of SceneDelegate?

Answer: SceneDelegate is used to manage the lifecycle of a single scene in your app, including its creation and destruction.

18. What is a Tuple and where should you use it?

Answer: A tuple is a group of multiple values that can be of different types. They are useful for returning multiple values from a function.

19. What are higher-order functions?

Answer: Higher-order functions are functions that take other functions as parameters or return functions as their result.

20. Create an array of strings ["1", "3", "4", "6"] and find the sum of all even numbers using higher-order functions. Don’t use any loops!

Answer:

```swift
let numbers = ["1", "3", "4", "6"]
let evenSum = numbers.compactMap { Int($0) }.filter { $0 % 2 == 0 }.reduce(0, +)
```

21. Create an array of Employees and try to filter out those who joined before Jan 2024.

Answer:

```swift
struct Employee {
    var name: String
    var joiningDate: Date
}
```

```
let employees: [Employee] = [] // Assume this array is populated
let filteredEmployees = employees.filter { $0.joiningDate <
Calendar.current.date(from: DateComponents(year: 2024, month: 1, day: 1))! }
```

22. What is Responder and what is its use?

Answer: The Responder chain is a hierarchy of objects that can respond to events and handle user interactions in iOS.

23. What are the different options you have to store data locally?

Answer: Options for local data storage include UserDefaults, Keychain, Core Data, and file storage.

24. What is the latest version of Xcode, Swift, iOS, and macOS?

Answer: (Provide the current versions at the time of the interview)

25. Do you know about WWDC?

Answer: : Yes, WWDC (Worldwide Developers Conference) is an annual event held by Apple to showcase new software and technologies for developers.

26. What is iOS and how is it different from other mobile OS?

Answer: iOS is Apple's mobile operating system, known for its user-friendly interface, security, and extensive ecosystem. It differs from Android and other OSes in its closed ecosystem and proprietary nature.

27. If your application is crashing, what are the different ways you can investigate to find out the issue?

Answer: Use Xcode debugger, check crash logs, utilize instruments to track memory usage, and analyze code for potential issues.

28. What is Jailbreak and what is its impact?

Answer: Jailbreaking is the process of removing software restrictions on iOS devices, allowing users to install unauthorized apps. It can compromise security and void warranties.

29. What frameworks are available to create UI in iOS?

Answer: Common frameworks include UIKit, SwiftUI, and Interface Builder.

30. What is the difference between UIKit and SwiftUI?

Answer: UIKit is the traditional framework for building iOS apps, while SwiftUI is a newer, declarative framework that allows for faster development and less boilerplate code.

31. What is an extension?

Answer: An extension allows you to add functionality to existing classes, structures, or enumerations without modifying their original implementation.

32. Use this link: https://hp-api.onrender.com/ to get the data from the server and parse it in a playground to print it.

Answer:

```swift
import Foundation

let url = URL(string: "https://hp-api.onrender.com/")!
let task = URLSession.shared.dataTask(with: url) { data, response, error in
    guard let data = data, error == nil else { return }
    // Parse data
}
task.resume()
```

33. What is Keychain? How will you decide whether to save data into Keychain or UserDefaults?

Answer: Keychain is a secure storage solution for sensitive data, while UserDefaults is used for storing user preferences. Use Keychain for sensitive information like passwords, and UserDefaults for simple user settings.

34. What is the problem with global variables?

Answer: Global variables can lead to namespace pollution, make code harder to test, and introduce unexpected side effects due to shared state.

35. What is Property Observer? Write an example.

Answer: Property observers allow you to respond to changes in a property’s value. Example:

```swift
var observedProperty: Int = 0 {
    willSet {
        print("About to set: \(newValue)")
    }
    didSet {
        print("Just set: \(oldValue) to \(observedProperty)")
    }
}
```

36. What is Property Wrapper? Write code to showcase it.

Answer: A property wrapper is a custom type that encapsulates the storage and behavior of a property. Example:

```swift
@propertyWrapper
struct Clamped<Value: Comparable> {
    private var value: Value
    private let range: ClosedRange<Value>

    var wrappedValue: Value {
        get { value }
        set { value = min(max(newValue, range.lowerBound), range.upperBound) }
    }
    init(wrappedValue: Value, _ range: ClosedRange<Value>) {
        self
```

37. Question: What sorting algorithm does the sorted() function in Swift use, and what is its time complexity in the average and worst-case scenarios?

Answer: The sorted() function in Swift uses the Timsort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort. Its time complexity is as follows:

Average Case: O(n log n) Worst Case: O(n log n) Timsort is designed to perform well on many kinds of real-world data and is particularly efficient for data that is already partially sorted.

38. Question: Given the following code snippets, which approach is safer for ensuring main-thread execution, and why?

```
// Approach 1
DispatchQueue.main.async {
    // UI update code
}

// Approach 2
DispatchQueue.main.sync {
    // UI update code
}
```

Answer:

- Approach 1 (DispatchQueue.main.async) is generally safer for ensuring main-thread execution, especially when called from a background thread. Using async avoids potential deadlocks, as it schedules the task to run on the main thread without blocking the calling thread.
- n Approach 2, where DispatchQueue.main.sync is used, you're telling the main thread: "Wait until this code is finished before moving on." This can be risky if you're already on the main thread, because the main thread is already busy running whatever code you just told it to wait on. Since it can't move on until that code is done, and the code can't start because the main thread is waiting, both get stuck waiting for each other. This creates a deadlock, where nothing can proceed.

39. Question: How would you ensure that a background task runs asynchronously while avoiding any blocking on the main thread?

Answer:

- Use DispatchQueue.global().async to run tasks on a background thread asynchronously. This prevents any blocking on the main thread, allowing UI updates or user interactions to remain responsive.

40. Question: Question: What’s the difference between DispatchQueue.main.async and DispatchQueue.global().async when updating UI elements?

Answer:

- DispatchQueue.main.async ensures code runs on the main thread, which is necessary for UI updates in iOS. DispatchQueue.global().async runs code on a background thread, which is ideal for tasks that don't involve UI changes or don’t require main-thread access, such as data processing or network requests.

40. Question: When should you prefer DispatchQueue.main.async over DispatchQueue.main.sync?

Answer:

- you should generally prefer DispatchQueue.main.async when updating the UI or performing tasks on the main thread from a background thread. Using async avoids potential deadlocks and doesn’t block the calling thread, ensuring smoother and safer execution on the main thread. sync is rarely used in this context as it can lead to deadlocks if misused.

41. Question: Explain a scenario where using DispatchQueue.global().asyncAfter might be helpful, and describe how it works.

Answer:

- DispatchQueue.global().asyncAfter is useful for delaying the execution of background tasks. For example, it might be used to implement a delay before retrying a failed network request. This method schedules code to run on a background queue after a specified delay without blocking the main thread, ensuring a responsive user interface.

42. In what scenario would DispatchQueue.main.sync be useful, and why is it generally avoided?

Answer:

- DispatchQueue.main.sync is rarely used in iOS development because it can easily lead to deadlock if called from the main thread. However, there are limited cases where it can be useful—typically, when you need to perform a task on the main thread immediately and you’re calling it from a background thread.

- For example, if you’re performing some heavy data processing in the background and need to update a critical UI element synchronously (such as disabling a button or updating a label instantly), DispatchQueue.main.sync ensures that the UI update happens right away. However, this should be used with caution and only from a background thread.

```
DispatchQueue.global().async {
  // Heavy data processing on a background thread
  let result = processData()

  // Synchronously updating a critical UI element on the main thread
  DispatchQueue.main.sync {
      self.resultLabel.text = "Result: \(result)"
      self.updateButton.isEnabled = false
  }
}
```

43. What is a non-escaping closure, and when would you typically use one in Swift?

Answer:

- A non-escaping closure is a closure that is guaranteed to be executed before the function it’s passed to returns. This means the closure does not “escape” or outlive the scope of the function. Non-escaping closures are the default in Swift.
- You typically use non-escaping closures when you need a closure to perform some work within the function itself and don’t need to store it to be used later. For example, you’d use a non-escaping closure for simple data processing tasks where the closure’s work can be completed within the function’s scope.

```swift
func performTask(_ task: () -> Void) {
  print("Start Task")
  task()  // The closure is executed within the function scope
  print("End Task")
}
```

performTask { print("Performing task") }

44. Why is it safer to use non-escaping closures when you don’t need the closure
to persist outside the function?
Answer:
- Non-escaping closures are safer because they avoid potential memory management
issues, like retain cycles. Since they do not outlive the function, the function
controls the closure’s lifecycle, making memory management straightforward. By
default, Swift treats closures as non-escaping, which helps prevent memory leaks
unless explicitly specified.

45. Give an example of when using an escaping closure is required instead of a
non-escaping closure.
Answer:
- An escaping closure is required when you need to call the closure after the function
has returned, such as in asynchronous operations.
- For instance, if you are fetching data from a server, the network request runs
asynchronously, so the closure must “escape” to be called once the request is
complete:
```swift
func fetchDataFromServer(completion: @escaping (Data?) -> Void) {
    DispatchQueue.global().async {
        // Simulate a network delay
        let data: Data? = Data()  // Simulated data
        completion(data)  // Called after function returns
    }
}
```

46. What are the different quality of service (QoS) classes in GCD, and which has the highest priority?

Answer:

- GCD defines several quality of service (QoS) classes to prioritize tasks based on their urgency and intended purpose. The classes are:
- User-interactive – Highest priority
- User-initiated
- Default
- Utility
- Background – Lowest priority
- The User-interactive class has the highest priority. It is used for tasks that need to update the UI immediately, such as responding to user actions. - - - - Conversely, the Background class has the lowest priority and is intended for tasks that are not time-sensitive, like pre-fetching data or syncing in the background.

47. When would you use the User-initiated QoS in GCD?

Answer:

- The User-initiated QoS class is used for tasks that the user expects to complete as part of their current interaction but are not necessarily required for immediate UI updates. It is useful for tasks that are started by a user action and should finish as soon as possible to provide a smooth user experience.

- For example, if a user selects an item to view details that require data loading, you would use the User-initiated QoS to load the data in the background, ensuring it completes promptly without blocking the main thread.

  ```swift
  DispatchQueue.global(qos: .userInitiated).async {
    // Fetch data needed for displaying item details
    let data = fetchData()
    DispatchQueue.main.async {
        // Update UI with data
        self.updateUI(with: data)
    }
  }
  ```

48. What is the Utility QoS, and when should it be used?

- Answer:*

- The Utility QoS class is used for tasks that the user is aware of but doesn't need immediately. This priority level is appropriate for tasks that involve long-running work, such as downloading files or processing data in the background. Tasks with this priority consume fewer system resources, preserving battery life and performance for higher-priority tasks.

```swift
DispatchQueue.global(qos: .utility).async {
  // Long-running task, such as downloading a large file
  let fileData = downloadLargeFile()
  DispatchQueue.main.async {
      // Update UI with completion status
      self.showDownloadComplete()
  }
}
```

49 .How does the Background QoS class differ from the Utility QoS class?

- Answer:*

- The Background QoS class is even lower in priority than Utility and is used for tasks that the user doesn't need to be aware of, such as prefetching data, indexing, or syncing. Background tasks are intended to run with minimal impact on system performance and battery life.

- Use Background QoS for work that can happen "behind the scenes" without the user noticing, like updating caches or synchronizing with a server.

50. Which GCD QoS class would you use for tasks that must respond to user actions in real-time, and why?

- Answer: For tasks that need to respond to user actions in real-time, you should use the User-interactive QoS class. This is the highest-priority class, ensuring tasks are performed immediately, typically on the main thread, for an instantaneous response to user interactions, such as animations or touch responses.

```swift
DispatchQueue.global(qos: .userInteractive).async {
  // Immediate response task, such as a small animation or UI update
  performQuickTask()
}
```

50 . GCD QoS Classes with Priorities:

- User-interactive – Immediate, time-sensitive tasks (highest priority).
- User-initiated – Tasks initiated by the user that should finish quickly.
- Default – The standard priority for tasks without a specified priority.
- Utility – Long-running tasks that do not need immediate attention.
- Background – Low-priority tasks that run behind the scenes (lowest priority).

51. Why Weak is slower than Strong ?

- refer to my medium article for this : https://medium.com/@shobhakartiwari/why-weak-is-slower-than-strong-e6c805784ed8

52. What is the difference between "immutable" and "mutable" objects in Swift?

- refer to my medium article for this : https://medium.com/@shobhakartiwari/immutable-vs-mutable-objects-in-swift-an-ios-developers-perspective-fae2763be87f

Swift Testing Interview Questions and Answers

53: What is unit testing, and why is it important?

A: Unit testing is a software testing method where individual components or functions are tested in isolation. It ensures code correctness, simplifies debugging, and enhances code quality.

:54 How do you set up a unit test target in Xcode?

A:

1. Open your Xcode project.
2. Select **File > New > Target**.
3. Choose **Unit Testing Bundle**.
4. Name the target and click **Finish**.

55: What is XCTest?

A: XCTest is Apple's testing framework used for writing unit tests and UI tests for iOS, macOS, watchOS, and tvOS applications.

56: How do you create a test case class using XCTest?

```swift
import XCTest

class MyTests: XCTestCase {
    func testExample() {
        let result = 2 + 2
        XCTAssertEqual(result, 4, "The result should be 4")
    }
}
```

56: Explain the lifecycle of a test method in XCTest.

A: XCTest provides the following lifecycle methods:

- `setUp()` : Called before each test method.
- `tearDown()` : Called after each test method.
- `setUpWithError()` & `tearDownWithError()` : Handle throwing errors if needed.

57: What are assertions in unit testing? Provide examples.

A: Assertions check test expectations. Common assertions:

```swift
XCTAssert(true)                // General assertion
XCTAssertEqual(2 + 2, 4)      // Equality check
XCTAssertNotEqual(2 + 2, 5)   // Inequality check
XCTAssertNil(nil)             // Nil check
XCTAssertNotNil("Hello")     // Non-nil check
```

58: How do you write asynchronous test cases using XCTest?

```
func testAsyncCall() {
    let expectation = self.expectation(description: "Async Call")

    fetchData { data in
        XCTAssertNotNil(data)
        expectation.fulfill()
    }

    waitForExpectations(timeout: 5, handler: nil)
}
```

59: What are test doubles? Explain mocks, stubs, and fakes.

A:

- **Mocks:** Objects that verify interactions.
- **Stubs:** Provide predefined responses.
- **Fakes:** Provide working implementations, usually for testing only.

60: How would you mock a network call in a unit test?

```
class MockNetworkService: NetworkService {
    func fetchData(completion: @escaping (Data?) -> Void) {
        let mockData = Data([0x0, 0x1, 0x2])
        completion(mockData)
    }
}
```

61: What is TDD, and what are its benefits?

A: TDD involves writing tests before writing the actual implementation code. Benefits include:

- Reduces bugs.
- Improves design.
- Provides better test coverage.

62: How would you follow TDD in Swift?

1. Write a failing test.
2. Implement the code to pass the test.
3. Refactor the code.

Example:

```
class CalculatorTests: XCTestCase {
    func testAddition() {
        let result = Calculator.add(2, 3)
        XCTAssertEqual(result, 5)
    }
}

struct Calculator {
    static func add(_ a: Int, _ b: Int) -> Int {
        return a + b
    }
}
```

63: What is code coverage, and how do you enable it in Xcode?

A: Code coverage measures how much of the source code is tested by unit tests.

To enable it:

1. Open Xcode.
2. Go to **Product > Scheme > Edit Scheme...**
3. Select **Test** and check **Gather Coverage for Targets**.

64: How do you interpret code coverage reports?

A: Xcode's code coverage tool highlights lines of code that are tested. Green means fully tested, while red indicates untested code.

64: What are best practices for writing unit tests?

- Write independent tests.
- Use meaningful test method names.
- Test one thing per method.
- Use test doubles when needed.
- Keep tests readable and maintainable.

65: How can you prevent unreliable tests from occurring?

- Avoid external dependencies.
- Use mock services.
- Ensure consistent test data.

66: How do you test Core Data models in unit tests?

```swift
func testCoreDataModel() {
    let context = PersistenceController.shared.container.viewContext
    let entity = MyEntity(context: context)
    entity.name = "Test"

    do {
        try context.save()
        XCTAssertNotNil(entity)
    } catch {
        XCTFail("Saving failed")
    }
}
```

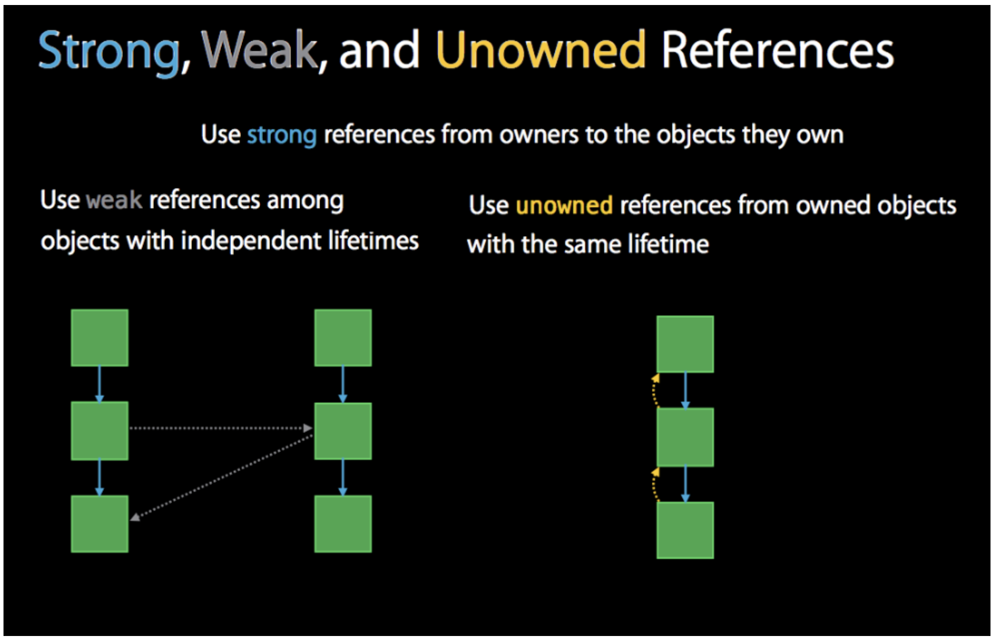
67: How do you test view models in MVVM architecture?

```swift
class ViewModelTests: XCTestCase {
    func testViewModelFetch() {
        let viewModel = MyViewModel(service: MockService())
        viewModel.fetchData()
        XCTAssertEqual(viewModel.data.count, 3)
    }
}
```

68. When you should use strong, weak and unowned?

- When working with reference types in Swift, managing memory is crucial to avoid retain cycles. Here's a quick guide to determine when to use □□□□□□, □□□□, or □□□□□□□ references:

□ □□□□□□ – Best for hierarchical relationships (e.g., parent owns child). □ □□□□ – Use when instances are optionally related to each other. □ □□□□□□□ – Use when one instance cannot exist without the other (mandatory dependency).



Strong, Weak, and Unowned References

Use strong references from owners to the objects they own

Use weak references among objects with independent lifetimes

Use unowned references from owned objects with the same lifetime