FIFTH EDITION

YOUR KEY TO UNLOCKING A NEW CAREER

# iOS Interview Handbook

A comprehensive guide to crack an iOS interview with top interview questions with answers, a roadmap for mastering your interview preparation.

Written By **Swiftable**

*Dear Reader,*

*The iOS job market in current time is not the same one you joined years ago.*
*Layoffs are common, hiring bars are higher, AI is reshaping what "**good enough**" means,*
*and every year a new wave of sharper, hungrier developers enters the field.*
*Your years of experience guarantee nothing, your comfort zone is a slow trap,*
*and the gap between those who keep learning and those who stop is widening every quarter.*

***Drop the excuses.** Not tomorrow, not next month, right now.*
*The developer you need to become in the next years will not be built by*
*the habits that got you here.*

## Table of Contents

# INTRODUCTION

## iOS Interview Handbook (Your key to unlocking a new career)

In today's competitive job market, having access to quality questions and a well-defined roadmap can give you a significant advantage over your peers. It equips you with the tools and knowledge needed to stand out during the interview process, increasing your chances of securing a good job.

**Interview Questions:** Dive into an extensive collection of 340+ curated iOS interview questions, meticulously selected to cover important topics and difficulty levels.

**Preparation Roadmap:** Navigate your journey to interview success with our comprehensive roadmap, meticulously crafted to provide you with a clear and structured path for interview preparation.

**Personalised Session:** Gain the exclusive opportunity to discuss your doubts in a personalised one-to-one session, where you'll receive tailored guidance, feedback, and strategies from an experienced iOS expert.

In future updates, the goal is to transform this book into the ultimate guide for iOS developers of all levels, from junior to senior. It will offer comprehensive guidance tailored specifically for interview preparation.

## Your Feedback Matters!

Your success in iOS interviews is our mission, and your insights help us make this handbook even better. We've poured our expertise into these 340+ questions and answers, but we know there's always room to grow and that's where you come in.

Drop us a line at devswiftable@gmail.com when:

**Got Questions?** Stuck on an answer or need clarification? Don't let doubts linger and shoot us an email and we'll help you out.

**Spotted an Error?** Found a typo, technical mistake, or something that doesn't quite add up? Let us know! Your eagle eyes help us maintain the quality everyone deserves.

**Ready for Your 1:1 Session?** Every reader gets a complimentary mentoring session! Email us to schedule yours, and we'll send you the meeting link. Let's discuss your interview prep, review tricky concepts, or strategise your career move.

**Have a Brilliant Idea?** Think we should add a particular topic? Have suggestions to make this handbook more valuable? Share your thoughts with us. If we implement your suggestion in the next version, we'll reward you with an **additional bonus 1:1 session** as our thank you!

We're not just authors. We’re your interview prep partners.

Chapter 01

# INTERVIEW PREPARATION ROADMAP

A focused, intensive plan for mid-to-senior iOS developers preparing for interviews. This roadmap maps directly to the chapters in this edition and assumes **4 to 5 hours of daily study**, with weekends reserved for problem-solving, catch-up, and revision.

*The next few years will not be kind to developers who coast on past experience. Drop the excuses, commit to your preparation roadmap, and outwork the version of you that has been postponing this.*

## How to Use This Roadmap

This roadmap is designed for a **45-day intensive preparation** with approximately **4 to 5 hours of daily study**. The schedule balances core iOS topics with two parallel tracks (Mobile System Design and Problem Solving / DSA) so you are not cramming them at the end.

### Time Allocation: Monday to Friday (~4 hours/day):

### Time Allocation: Saturday & Sunday (~5 to 6 hours/day):

### Practical Approach While Preparing

### Mock Interview Strategy

*Read the full version of the book to see this…*

## Phase 0: Self-Assessment (Day 0 - Before You Start)

## Week 1: Swift Foundations & OOP

**System Design focus this week:** News Feed Design

**DSA focus this week:** Arrays, Strings, Two Pointers, Sliding Window

The goal of this week is to lock down language fundamentals. Interviewers test these early to gauge depth.

**Day 1 - Class, Structure, Actors & Enumeration:** Value vs reference types, COW, deep vs shallow copy, recursive enums, actors basics. **System Design:** News Feed requirements and scope.

**Day 2 - Properties & Initializers:** Stored vs computed, lazy, property wrappers, designated/convenience/failable/required initializers. **DSA:** Array fundamentals (in-place modification, prefix sums, basic patterns).

**Day 3 - Functions, Methods & Closures:** Higher-order functions, escaping/non-escaping, capture lists, autoclosures, retain cycles in closures. **System Design:** News Feed HLD (feed service, ranking, pagination).

**Day 4 - Protocol & Delegation:** Protocol extensions, composition, associated types, method dispatch in protocols, PATs vs generics. **DSA:** String fundamentals (character manipulation, palindrome patterns).

**Day 5 - Generics & Error Handling:** Type constraints, type erasure, where clauses, custom errors, throws/rethrows, async error handling. **System Design:** News Feed LLD (data models, caching layer, infinite scroll).

**Day 6 - DSA + Catch-up:** Problem solving on mixing Arrays and Strings. Catch up on any rushed topic from this week.

**Day 7 - Weekly Revision + Mock Interview #1:** Complete revision of this week. Self or peer mock interview.

## Week 2: Memory & Concurrency

## Week 3: Networking, Combine & UIViewController

## Week 4: UIKit (Continue) & SwiftUI

## Week 5: Architecture, SOLID & Performance

## Week 6: Security, Persistence & Final Polish

## Week 7: Miscellaneous, Live Coding & Game Day

## Beyond Day 45: Continuing the Journey

## Suggested Side Projects (Optional, If You Have Buffer Time)

*Read the full version of the book to see this…*

Chapter 02

# CLASS, STRUCTURE, ACTORS & ENUMERATION

## Q. What are the differences between class and structure?

Class and Structure both are used for creating custom data types. While they share similarities with their counterparts, there are some specific differences and considerations you should know. To understand the differences between them, we will use the below example:

```
class MediaAssetClass {
    var name: String
    var type: String
    init(name: String, type: String) {
        self.name = name
        self.type = type
    }
}

struct MediaAssetStruct {
    var name: String
    var type: String
}
```

### Reference Types Vs. Value Types

When you assign an instance of a class to a variable or pass it as an argument to a function, you're working with a reference to the original instance. Any changes made to that reference affect the original instance. For example:

```
var jpgMedia = MediaAssetClass(name: "ProfilePhoto_123", type: "JPG")
var jpgMediaDuplicate = jpgMedia
// creating a reference to the original instance

jpgMediaDuplicate.name = "Profile_123"
print(jpgMedia.name) // Print: Profile_123
print(jpgMediaDuplicate.name) // Print: Profile_123
```

In the above example, the jpgMedia and jpgMediaDuplicate are references to the same instance. When you modify the name property of jpgMediaDuplicate, it also changes the name property of the original jpgMedia instance.

When you assign an instance of a structure to a variable or pass it as an argument to a function, you're working with a copy of the original instance. Changes made to the copy do not affect the original instance unless explicitly mutated using the mutating keyword. For example:

```
var movMedia = MediaAssetStruct(name: "Video_123", type: "MOV")
var movMediaDuplicate = movMedia
// creating a copy of the original instance

movMediaDuplicate.name = "VideoFile_123"
print(movMedia.name) // Print: Video_123
print(movMediaDuplicate.name) // Print: VideoFile_123
```

In the above example, movMedia and movMediaDuplicate are separate instances. When you modify the name property of movMediaDuplicate, it does not affect the original movMedia instance.

### Inheritance

Classes support inheritance, allowing one class to inherit properties and methods from another class. While, structure doesn't support inheritance. You cannot subclass a structure. For example:

```
// attempting to define a struct that inherits from another struct - This will result in a compilation error.
struct PhotoAssetStruct: MediaAssetStruct {}

// Compilation Error: 'Inheritance from non-protocol, non-class type 'MediaAssetStruct''
```

If you need to achieve similar behaviour to inheritance with structs, you can use protocols and protocol extensions, but this would not be true inheritance.

### Identity Checking

Classes have identity, and you can check if two references point to the same instance using the `===` operator. For example:

```
var jpgMedia = MediaAssetClass(name: "ProfilePhoto_123", type: "JPG")
var jpgMediaDuplicate = jpgMedia
jpgMediaDuplicate.name = "Profile_123"

if jpgMedia === jpgMediaDuplicate {
    print("Both class objects point to the same instance.")
} else {
    print("Both class objects do not point to the same instance.")
}
// Prints: Both class objects point to the same instance.
```

Structure do not have identity checks like classes. You compare instances of struct by comparing their properties. For example:

The `==` operator is not automatically defined for structs. Therefore, we need to explicitly define how to compare instances of our custom struct.

```
var movMedia = MediaAssetStruct(name: "Video_123", type: "MOV")
var movMediaDuplicate = movMedia
movMediaDuplicate.name = "VideoFile_123"

// error: binary operator '==' cannot be applied to two 'MediaAssetStruct' operands
if movMedia == movMediaDuplicate {
    print("Both struct objects have the same properties.")
} else {
    print("Both struct objects do not have the same properties.")
}
```

Let's correct the example by implementing the Equatable protocol:

```
struct MediaAssetStruct: Equatable {
    var name: String
    var type: String
}

// Run the above example now and you will see the output like:
// Both struct objects do not have the same properties.
```

By conforming to Equatable, the compiler will compare all the properties of both the instances. In case of custom comparison with Equatable protocol, you can override `static ==` function.

## Immutability

Instances of classes can have mutable properties, and you can modify these properties even if the class instance is declared as a constant (using `let`). By default, instances of struct are immutable (constants). To modify the properties, you need to mark the method that performs the modification with the mutating keyword. For example:

```
struct MediaAssetStruct: Equatable {
    var name: String
    var type: String

    // error: mark method 'mutating' to make 'self' mutable
    func modifyName(newName: String) {
        self.name = newName
    }
}
```

## Deinitializers

In classes, deinitializers are called immediately before an instance of the class is deallocated. Deinitializers can also access properties and other members of the class instance and can perform any cleanup necessary for those members. For example:

```
class MediaAssetClass {
    deinit {
        print("class instance is deallocated.")
    }
}

var jpgMedia: MediaAssetClass? = MediaAssetClass()
jpgMedia = nil
// Prints: class instance is deallocated.
```

Because structs are value types and are copied when passed around, there's no concept of deinitializing an instance of a struct in the same way as with classes. For example:

```
// error: deinitializers may only be declared within a class, actor, or non-copyable type
struct MediaAssetStruct {
    deinit {
        print("struct instance is deallocated.")
    }
}
```

In summary, you should consider the differences between both based on reference vs. value semantics, inheritance, immutability, deinitlaization etc.

## Q. How are actors different from classes and structures?

Actors were introduced in Swift 5.5 as a concurrency primitive. They are a third concrete nominal type alongside classes and structures, specifically designed to protect mutable state from data races.

**Reference vs value semantics:** Actors are reference types (like classes), so they're suitable for sharing state. Structures are value types. This is an important clarification: actors are effectively "classes with built-in isolation."

**Data-race safety:** Actors encapsulate their mutable state and serialize access to it automatically, eliminating data races at the language level. Classes and structures provide no such guarantee. Safe concurrent access with a class typically requires manual synchronization such as locks, serial queues, or barriers.

**Access rules:** When you touch an actor's mutable state or call its methods from outside, you must do so asynchronously using await. From inside the actor (via self), access is synchronous and no await is needed. Classes and structures have no such rule; any caller can touch their state directly.

```
actor Counter {
    private var value = 0
    func increment() { value += 1 } // synchronous inside
```

```
    func current() → Int { value }
}

let counter = Counter()
await counter.increment() // await required from outside
let n = await counter.current()
```

**Inheritance:** Actors do not support inheritance or subclassing, so override and final don't apply to them. The one exception is that an actor may inherit from NSObject for Objective-C interoperability. In effect, an actor behaves like a final class.

**Initializers:** Because actors don't support inheritance, their initializer rules are simpler than classes. Actors don't use the convenience keyword; a delegating initializer is simply one that calls 'self.init(…)', the same rule that applies to structs. Classes are the only type that uses the designated/convenience distinction, which exists precisely to manage inheritance.

**Re-entrancy:** Actors are re-entrant. When a method hits an await suspension point, the actor can service other messages in the meantime. This prevents deadlocks but means you cannot assume the actor's state is unchanged across an await. Re-check invariants after suspension. Classes and structs have no such consideration because they don't have an execution model.

### How to decide?

- Reach for a **struct** as the default. It gives you value semantics, implicit Sendable in most cases, and no concurrency concerns to reason about.
- Reach for a **class** when you need inheritance, Objective-C interoperability, a deinit, or shared mutable state that is already confined to a single thread or guarded by your own synchronization.
- Reach for an **actor** when you need shared mutable state that will be accessed from multiple concurrency contexts. The actor gives you that sharing safely, without hand-rolled locks, and integrates cleanly with async/await.

Actors provide a higher-level abstraction for concurrent programming compared to classes and structures, with built-in support for safe, asynchronous messaging and encapsulation of mutable state.

*Read the full version of the book to see this…*

Chapter 03

# PROPERTIES & INITIALIZERS

## Q. What are property observers and when can they be useful?

Property observers allow you to observe and respond to changes in the value of a property. There are two types of property observers available: willSet and didSet.

**willSet:** This observer is called just before the value of the property is set. It provides the new value as a constant parameter, which you can use to perform actions before the value is updated.

**didSet:** This observer is called immediately after the value of the property is set. It provides the old value of the property as a constant parameter, which you can use to perform actions after the value has been updated.

```
struct MediaAssetStruct {
    var name: String {
        willSet {
            print("About to change name to \(newValue)")
        }
        didSet {
            print("Name changed from \(oldValue) to \(name)")
        }
    }

    var size: Int {
        didSet {
            if size > 100 {
                print("Warning: File size is large!")
            }
        }
    }
}

var mediaAsset = MediaAssetStruct(name: "ProfilePhoto", size: 50)
mediaAsset.name = "NewProfilePhoto"
// Prints: About to change name to NewProfilePhoto
// Prints: Name changed from ProfilePhoto to NewProfilePhoto

mediaAsset.size = 120
// Prints: Warning: File size is large!

var mediaAsset = MediaAssetStruct(name: "ProfilePhoto", size: 50)
mediaAsset.name = "NewProfilePhoto"
// Prints: About to change name to NewProfilePhoto
// Prints: Name changed from ProfilePhoto to NewProfilePhoto

mediaAsset.size = 120
// Prints: Warning: File size is large!
```

In the example, property observers are used to print messages before and after changing the property name, and to print a warning message if the size exceeds a certain threshold. These observers help in maintaining the integrity of the properties and executing additional logic when they are modified.

*Note that property observers are not triggered when a property is set within its own observer. This prevents infinite recursion and ensures predictable behavior.*

**They are useful in various scenarios:**

Validation: You can use property observers to enforce validation rules on property values. For example, you can ensure that a temperature value stays within a certain range.

Updating UI: They are commonly used to update the user interface in response to changes in property values. For instance, you might update a label's text when a related property changes.

Logging and Debugging: They can be helpful for logging and debugging purposes. You can use them to log property changes or track down bugs related to property values.

Side Effects: They allow you to encapsulate side effects related to property changes within the property itself, improving code readability and maintainability.

## Q. What are the designated initializers? Can a class or struct have multiple designated initializers?

Designated initializers are the primary initializers for a class or struct. They are responsible for initializing all properties introduced by that class or struct and ensuring that the instance is fully initialized before it's used.

A designated initializer is marked with the init keyword, and it must initialize all properties introduced by that class or struct, either by assigning initial values directly or by calling other initializers.

A class or struct can have multiple designated initializers, each of which initializes a subset of properties or provides different initialization paths. These multiple designated initializers can have distinct parameter lists and initialization logic, but they all must ensure that all properties are initialized before the instance is considered fully initialized. For example:

```
struct MediaAssetStruct {
    var name: String
    var type: String

    init(name: String, type: String) {
        self.name = name
        self.type = type
    }

    // additional designated initializer
    init(name: String) {
        self.init(name: name, type: "Unknown")
    }
}
```

```
let mediaAsset1 = MediaAssetStruct(name: "Photo", type: "JPEG")
let mediaAsset2 = MediaAssetStruct(name: "Video")

print("Name: \(mediaAsset1.name) and type: \(mediaAsset1.type)")
// Prints: Name: Photo and type: JPEG

print("Name: \(mediaAsset2.name) and type: \(mediaAsset2.type)")
// Prints: Name: Video and type: Unknown
```

In this example, MediaAssetStruct has two designated initializers. The first one initializes both name and type, while the second one initializes only name, setting type to a default value of “Unknown".

```
class MediaAssetClass {
    var name: String
    var type: String

    init(name: String, type: String) {
        self.name = name
        self.type = type
    }

    // additional designated initializer
    init(name: String) {
        self.name = name
        self.type = "Unknown"
    }
}

let mediaAsset3 = MediaAssetClass(name: "Audio", type: "MP3")
let mediaAsset4 = MediaAssetClass(name: "Document")

print("Name: \(mediaAsset3.name) and type: \(mediaAsset3.type)")
// Prints: Name: Audio and type: MP3

print("Name: \(mediaAsset4.name) and type: \(mediaAsset4.type)")
// Prints: Name: Document and type: Unknown
```

Similarly, MediaAssetClass also has two designated initializers. The first one initializes both name and type, while the second one initializes only name, setting type to a default value of "Unknown".

If you don't assign all properties in its initializer, you'll get a compiler error. It’s require that all properties have a value before the initializer completes its execution. This ensures that an instance of the struct is always in a valid state.

```
struct MediaAssetStruct {
    var name: String
```

```
    var type: String

    init(name: String, type: String) {
        self.name = name
        self.type = type
    }

    // additional designated initializer
    init(name: String) {
        // self.init(name: name, type: "Unknown")
        self.name = name
    }
}

// error: return from initializer without initializing all stored properties
```

Since type is not assigned in the initializer, the struct instance would be in an invalid state if it were allowed to be created.

*Read the full version of the book to see this…*

Chapter 04

# FUNCTIONS, METHODS & CLOSURES

## Q. What is the difference between escaping and non-escaping closures?

Closures can capture and store references to constants and variables from the context within which they are defined. These captured values can lead to a  reference cycle.  This is where the closure captures a reference to a value that also has a strong reference back to the closure, causing a memory leak. To avoid memory leaks, Swift provides two types of closure: escaping and non-escaping closures.

### Non-Escaping Closures

A non-escaping closure is guaranteed to be executed within the scope in which it is defined. This means the closure is invoked before the function containing it returns. The compiler knows that the closure won’t be used outside the function and optimize the code accordingly.

Non-escaping closures are the default behavior. These closures are typically used for synchronous operations within the function. You don't need to mark a closure as non-escaping explicitly; it's inferred by default. For example:

```
// non-escaping closure
func execute(closure: () → Void) {
    print("Executing non-escaping closure")
    closure()
```

```
    print("Finished executing non-escaping closure")
}

execute {
    print("This is a non-escaping closure")
}

// Prints:
// Executing non-escaping closure
// This is a non-escaping closure
// Finished executing non-escaping closure
```

In this example, the closure is called synchronously within the `execute` function.

### Escaping Closures

An escaping closure can be stored or called after the function that contains it has returned. Escaping closures are useful when you want to perform an operation asynchronously or store a closure for later use. To allow a closure to escape the function's scope, you must mark it explicitly using the `@escaping` keyword. For example:

```
var escapingClosureArray: [() → Void] = []

func addEscapingClosureToQueue(closure: @escaping () → Void) {
    print("Adding escaping closure to queue")
    escapingClosureArray.append(closure)
}

addEscapingClosureToQueue {
    print("This is an escaping closure - 1")
}

addEscapingClosureToQueue {
    print("This is an escaping closure - 2")
}

print("Before closure execution")
escapingClosureArray.forEach { $0() }
print("After closure execution")

// Prints:
// Adding escaping closure to queue
// Before closure execution
// This is an escaping closure - 1
// This is an escaping closure - 2
```

```
// After closure execution
```

In the above example:

- Here, escapingClosureArray is declared as an array of closures that take no arguments and return Void.
- The function addEscapingClosureToQueue takes an escaping closure as a parameter and appends it to the escapingClosureArray.
- Two escaping closures are added to the escapingClosureArray using the addEscapingClosureToQueue function.
- The output shows that the closures added to the array they retain their intended order of execution, reflecting the order in which they were added to the array.

*Read the full version of the book to see this…*

Chapter 05

# PROTOCOL & DELEGATION

## Q. What is a protocol composition? Can you explain how it's related to protocol extensions?

Protocol composition is a powerful feature that allows you to combine multiple protocols into a single name. This can be very useful when you want to define a type that needs to adhere to multiple protocols simultaneously. You can combine multiple protocols into a single name instead of writing them repeatedly.

You can compose protocols using the `&` operator. For example:

```
protocol Printable {
    func printDescription()
}

protocol Editable {
    func edit()
}

// protocol composition
typealias PrintableAndEditable = Printable & Editable
```

Now, any type that conforms to `PrintableAndEditable` must also conform to both `Printable` and `Editable`.

Protocol extensions allow you to provide default implementations for protocol methods. When combined with protocol composition, you can provide default implementations for methods required by multiple protocols.

```
extension Printable {
```

```
    func printDescription() {
        print("Printable description")
    }
}
extension Editable {
    func edit() {
        print("Editable content edited")
    }
}
```

Example of conforming to the composed protocol:

```
struct MediaAssetStruct: PrintableAndEditable {
    // add additional properties or methods here
    // no need to implement printDescription() and edit() methods
}

let mediaAsset = MediaAssetStruct()
mediaAsset.printDescription() // Prints: Printable description
mediaAsset.edit() // Prints: Editable content edited
```

In the above example, MediaAssetStruct conforms to the composed protocol `PrintableAndEditable`. Since both `Printable` and `Editable` have default implementations provided through protocol extensions, MediaAssetStruct automatically inherits these implementations without needing to define them explicitly. This makes the code cleaner and more maintainable.

*Read the full version of the book to see this…*

Chapter 20

# UNIT TESTING, XCTEST

## Q. What is the difference between XCTest and Swift Testing frameworks?

XCTest uses a traditional class-based approach where your test classes inherit from XCTestCase and test methods must start with the word "test".

Swift Testing is new testing framework. It was designed specifically for Swift from the ground up, taking advantage of modern Swift features like macros, async/await, and the type system. Instead of using classes and inheritance, it uses free functions annotated with the @Test macro.

The syntax differences are quite significant. With XCTest, you write:

```
class UserServiceTests: XCTestCase {
    var service: UserService!
```

```
    override func setUp() {
        super.setUp()
        service = UserService()
    }

    func testUserLogin() {
        let result = service.login(email: “devswiftable@gmail.com”)
        XCTAssertTrue(result)
    }
}
```

With Swift Testing, the same test looks like:

```
@Test func userLogin() {
    let service = UserService()
    let result = service.login(email: “devswiftable@gmail.com”)
    #expect(result == true)
}
```

Notice how Swift Testing is more concise. You don't need setUp or tearDown methods because you can just create instances in the test itself. The #expect macro is more natural to read than XCTAssert methods.

One of the biggest advantages of Swift Testing is **parameterised tests**. With XCTest, if you want to test the same logic with different inputs, you have to write separate test methods or loop through values in a single test. Swift Testing makes this elegant:

```
@Test(arguments: [("valid@email.com", true), ("invalid-email", false), ("", false)])
func emailValidation(email: String, expected: Bool) {
    let validator = EmailValidator()
    #expect(validator.isValid(email) == expected)
}
```

This runs as three separate tests in the test navigator, making failures easier to diagnose. With XCTest, you'd need to write three different test methods or use a loop that makes debugging harder.

Swift Testing also has a trait system that lets you **organize and control test behavior**. You can mark tests with .bug(id: "BUG_ID") to track which bug they're testing, .disabled("Waiting for backend API fix") to temporarily skip tests, or .serialized to prevent parallel execution for tests that can't run concurrently.

```
@Test(.bug("BUG_ID", "Login fails with special characters in password"))
func loginWithSpecialCharacters() {
    let service = UserService()
    #expect(service.login(email: “devswiftable@email.com", password: "p@$$w0rd!"))
}

@Test(.disabled("Waiting for backend API fix"))
```

```
func premiumSubscriptionValidation() {
    let service = SubscriptionService()
    #expect(service.isValid(userID: "user_46451"))
}
```

However, Swift Testing has limitations as it requires iOS 18+ versions. It also lacks some features XCTest has, like UI testing support (Swift Testing is only for unit tests), performance measurement APIs, and certain specialized assertion types.

Choose XCTest when you're working on an existing project with established XCTest suites, need to support older iOS versions, require UI testing capabilities, or when your team is already comfortable with the framework and sees no compelling reason to switch.

Choose Swift Testing when you want to leverage modern Swift features, need better parameterised testing, want more expressive test code, or when you can mandate iOS 18+ as your minimum deployment target.

## Q. How would you make a legacy class testable that has tight coupling and multiple dependencies?

Legacy code is often difficult to test because it was written before testing became a priority, or because shortcuts were taken under deadline pressure. The typical issues are tight coupling (creating dependencies inside the class), global state (singletons, static properties), and God objects (classes that do too much). Making such code testable without breaking existing functionality requires careful, incremental refactoring.

Let's start with a realistic example of untestable legacy code:

```
class OrderProcessor {
    func processOrder(orderId: String) → Bool {
        let database = DatabaseManager.shared
        let network = NetworkClient()
        let analytics = AnalyticsManager.shared
        let notifications = NotificationCenter.default

        guard let order = database.fetchOrder(id: orderId) else {
            return false
        }

        let paymentResult = network.processPayment(
            amount: order.total,
            card: order.paymentMethod
        )

        if paymentResult.success {
            database.updateOrderStatus(orderId, status: .completed)
            analytics.track("order_completed", properties: ["amount": order.total])
            notifications.post(name: .orderCompleted, object: order)
```

```
            return true
        }

        return false
    }
}
```

This class is impossible to test properly because it creates all its dependencies internally. You can't test payment processing without hitting a real network. You can't test database updates without a real database. Every test would have side effects.

The gold standard solution is dependency injection means passing dependencies through the initializer. But we need to do this without breaking existing code that's already calling this class. The trick is to use default parameters:

```
class OrderProcessor {
    private let database: DatabaseManager
    private let network: NetworkClient
    private let analytics: AnalyticsManager

    init(database: DatabaseManager = .shared,
         network: NetworkClient = NetworkClient(),
         analytics: AnalyticsManager = .shared) {
        self.database = database
        self.network = network
        self.analytics = analytics
    }

    func processOrder(orderId: String) -> Bool {
        guard let order = database.fetchOrder(id: orderId) else {
            return false
        }

        let paymentResult = network.processPayment(
            amount: order.total,
            card: order.paymentMethod
        )

        if paymentResult.success {
            database.updateOrderStatus(orderId, status: .completed)
            analytics.track("order_completed", properties: ["amount": order.total])
            return true
        }

        return false
```

```
    }
}
```

Existing code still works because OrderProcessor() uses the default singletons. But tests can now inject mocks:

```
func testSuccessfulOrderProcessing() {
    let mockDatabase = MockDatabaseManager()
    let mockNetwork = MockNetworkClient()
    let mockAnalytics = MockAnalyticsManager()

    let processor = OrderProcessor(
        database: mockDatabase,
        network: mockNetwork,
        analytics: mockAnalytics
    )

    mockDatabase.stubOrder = Order(id: "357", total: 99.99)
    mockNetwork.stubPaymentResult = PaymentResult(success: true)

    let result = processor.processOrder(orderId: "357")

    XCTAssertTrue(result)
    XCTAssertTrue(mockAnalytics.trackCalled)
}
```

But wait - the dependencies are concrete classes, not protocols, so we can't create mocks yet. We need to introduce protocols, which is called "Extract Interface" refactoring:

```
protocol DatabaseManaging {
    func fetchOrder(id: String) → Order?
    func updateOrderStatus(_ id: String, status: OrderStatus)
}

protocol NetworkProviding {
    func processPayment(amount: Double, card: String) → PaymentResult
}

protocol AnalyticsTracking {
    func track(_ event: String, properties: [String: Any])
}

// make existing classes conform
extension DatabaseManager: DatabaseManaging { }
```

```
extension NetworkClient: NetworkProviding { }
extension AnalyticsManager: AnalyticsTracking { }

// update OrderProcessor
class OrderProcessor {
    private let database: DatabaseManaging
    private let network: NetworkProviding
    private let analytics: AnalyticsTracking

    init(database: DatabaseManaging = DatabaseManager.shared,
         network: NetworkProviding = NetworkClient(),
         analytics: AnalyticsTracking = AnalyticsManager.shared) {
        self.database = database
        self.network = network
        self.analytics = analytics
    }
}
```

Now we can create test doubles that conform to these protocols. The production code is unchanged because the defaults still use the real implementations.

Sometimes legacy code does too much in one method. In that case, extract smaller methods first:

```
class OrderProcessor {
    func processOrder(orderId: String) -> Bool {
        guard let order = fetchOrder(orderId) else {
            return false
        }

        let paymentResult = processPayment(for: order)

        if paymentResult.success {
            completeOrder(orderId)
            trackCompletion(order)
            return true
        }

        return false
    }

    private func fetchOrder(_ id: String) -> Order? {
        return database.fetchOrder(id: id)
    }

    private func processPayment(for order: Order) -> PaymentResult {
```

```
        return network.processPayment(amount: order.total, card: order.paymentMethod)
    }

    private func completeOrder(_ id: String) {
        database.updateOrderStatus(id, status: .completed)
    }

    private func trackCompletion(_ order: Order) {
        analytics.track("order_completed", properties: ["amount": order.total])
    }
}
```

This makes the main method easier to understand and sets you up for further refactoring. Each small method can potentially be tested in isolation or extracted to its own class.

For really tangled code with multiple responsibilities, you might extract entire classes:

```
class PaymentProcessor {
    private let network: NetworkProviding

    init(network: NetworkProviding = NetworkClient()) {
        self.network = network
    }

    func process(order: Order) → PaymentResult {
        return network.processPayment(amount: order.total, card: order.paymentMethod)
    }
}

class OrderCompletionHandler {
    private let database: DatabaseManaging
    private let analytics: AnalyticsTracking

    func completeOrder(_ order: Order) {
        database.updateOrderStatus(order.id, status: .completed)
        analytics.track("order_completed", properties: ["amount": order.total])
    }
}
```

The key principles are: make small, safe changes; maintain backward compatibility; add tests as you refactor; and don't try to fix everything at once. Each small improvement makes the code slightly more testable. Over time, the codebase becomes easier to work with.

Chapter 23

# LIVE CODING ROUND

You can write code and solve complex problems efficiently. No doubts!

But it is important to know how much time you take to solve the problem and most important how you think while solving a problem (specially complex problems). Are you efficient to solve problem in a time manner?

Live coding interviews helps to evaluate how you approach and solve complex problems in real-time. Live coding requires you to think in a live environment, adapt to new challenges (e.g. edge cases), and demonstrate your logical reasoning and analytical skills under pressure.

An interviewer can directly assess your coding abilities, syntax knowledge, and understanding of core iOS concepts. This hands-on evaluation helps determine whether you possesses the technical skills necessary for the role you're applying for.

Another important aspect behind live coding rounds can include debugging tasks that assess your ability to identify, diagnose, and resolve errors efficiently. This evaluation helps determine how you approach debugging techniques in solving complex problems.

*It is easy to write good code, but it is difficult to solve problems in a timely manner. Every company seeks candidates who are logical thinkers and efficient problem solvers. You can work on problem-solving skills every day by practicing.*

## What to expect in a live coding round?

Navigating a live coding round can be both exciting and nerve-wracking for you. As this is an important interview round, you should understand **what to prepare** to enhance your preparedness and confidence. This section outlines the typical structure, environment, types of tasks, and key elements involved in a live coding interview for an iOS position. Let's understand different aspects to prepare for coding rounds.

**Duration:** Typically lasting between 45 minutes to an hour, allowing sufficient time to present and solve one or more problems.

**Platform:** Conducted either in-person using Xcode/Playgrounds or remotely via online coding platforms such as CoderPad, HackerRank, or custom company tools.

**Interaction:** A real-time interactive session where you shares the screen and collaborates with the interviewer.

**Problem Scope:** May involve solving algorithmic challenges, building small UI components, or debugging existing code snippets relevant to iOS development.

**Thought Process:** Might ask to articulate your reasoning, approach, and decision-making as you work through the problem.

*Remember, confidence is very important during this round. A good preparation will boost your confidence to performance awesome.*

### What is the general evaluation criteria?

You should understand how you will be assessed, so that you can focus on key areas during the interview:

- Correctness and Functionality: Ensuring that your solution works as intended and meets all specified cases and requirements.
- Quality and Readability: Writing clean, maintainable code with proper naming conventions, indentation, and modularisation.
- Optimization: Implementing solutions that are not only correct but also optimized for performance and resource usage.
- Approach: Showing a logical and structured approach to breaking down and tackling complex problems.
- iOS-Specific Knowledge: Applying relevant frameworks, design patterns, and best practices specific to iOS development.
- Adaptability: Handling unexpected challenges, adapting to feedback, and maintaining composure under pressure.

### Types of Live Coding Challenges

As live coding round is important part of the interview process, you should focus on various types of coding challenges. Because, understanding the various types of live coding challenges you may encounter can help you prepare effectively and showcase your problem solving abilities comprehensively.

*Read the full version of the book to see this…*

## iOS Interview Handbook (Your key to unlocking a new career)

Thank you for investing your time and trust in our interview preparation eBook. As you've navigated through the comprehensive content, we hope you've found valuable insights and resources to empower your journey towards interview success.

We're committed to continuous improvement, and your feedback plays a crucial role in shaping the future editions of this book. If you've encountered any errors, ambiguities, or have suggestions for enhancements, please don't hesitate to reach out to us via email. Your input is invaluable in our mission to provide the best possible resources for iOS developers.

Once again, thank you for choosing our eBook. We wish you the best of luck in your iOS interviews and future endeavours. Keep striving for excellence, and remember, your success is our success!

If you have any doubts or queries, please don't hesitate to reach out to us via email:
**devswiftable@gmail.com**

Best of luck!

— *Nitin A*