THE SENIOR IOS INTERVIEW GUIDE

A Deep, Practical Guide for iOS Developers

THINK • EXPLAIN • DECIDE

KIRAN JOTE
Lead iOS Developer, Expedia

AKASH SHARMA
Lead iOS Developer, Google

UNLOCK YOUR FULL POTENTIAL

Cracking the iOS Interview

A Deep, Practical Guide for iOS Developers

Kiran Jote

Author · Lead iOS Developer, Expedia

Akash Sharma

Co-author · Lead iOS Developer, Google

Copyright

Author

Kiran Jote
Lead iOS Developer, Expedia

Co-author

Akash Sharma
Lead iOS Developer, Google

License

Disclaimer

Version Information

Version: 1.0
Release Date: January 2026

Sample Preview

This is a **free sample** of *Cracking the iOS Interview*.

It contains the preface and Chapter 1 — Memory Management & ARC.

The complete book has 18 chapters covering Swift internals, concurrency, architecture, networking, SwiftUI, and interview strategy.

Full Contents

Included in this sample

In the full book

Preface

Preparing for iOS interviews often feels frustrating for experienced developers.

You may already know Swift, UIKit, SwiftUI, and the iOS ecosystem well, yet still struggle to articulate your thinking under interview pressure. Many candidates fail not because they lack knowledge, but because they are unable to clearly explain *why* a particular approach was chosen, what trade-offs were considered, or how a decision would scale in production.

This book was written to address that exact gap.

Rather than focusing on memorization or isolated facts, this guide emphasizes **reasoning, decision-making, and clear communication**—the skills interviewers actively look for when evaluating mid to senior iOS developers.

The chapters in this book are structured around the questions that are most frequently asked in real iOS interviews. Each topic is explained with practical context, common misconceptions, and the mental models used by experienced engineers. From Chapter 7 onward, you will also find concise follow-up interview questions with spoken-style answers to help you respond confidently in live interviews.

This is not a beginner's introduction to iOS or Swift. The book assumes that you have already built real applications and are now looking to refine how you reason about architecture, concurrency, performance, and system design.

You can read this book end-to-end or use it as a targeted reference while preparing for interviews. In both cases, the goal remains the same: to help you think and communicate like a strong iOS engineer in high-stakes interview situations.

Chapter 1: Memory Management & ARC in iOS

Interview Question

How does memory management work in iOS?

Why Interviewers Ask This

This question tests whether you understand how objects are created, retained, and destroyed in memory. Interviewers use this to evaluate whether you can write **safe, leak-free production code**.

At senior levels, weak understanding of memory management is a strong rejection signal.

Ideal Interview Answer (2–3 Minutes)

iOS uses **Automatic Reference Counting (ARC)** to manage memory. ARC keeps track of the number of **strong references** pointing to an object. When the reference count reaches zero, the object is deallocated automatically.

Developers do not manually free memory, but they must avoid **retain cycles** by using `weak` or `unowned` references appropriately.

Deep Explanation

What ARC Really Is

ARC is a **compile-time memory management system**, not a runtime garbage collector.

The Swift compiler automatically inserts: - `retain` - `release`

instructions during compilation.

Because ARC works at compile time: - There are no garbage collection pauses - Memory behavior is predictable - UI performance remains smooth

ARC **does not prevent memory leaks**. It only automates reference counting.

Strong References

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

var user: User? = User(name: "Kiran")
```

- `user` holds a strong reference
- Reference count = 1
- Object remains alive

When:

```
user = nil
```

- Reference count becomes 0
- `deinit` is called
- Memory is released

Weak References

```
weak var delegate: SomeDelegate?
```

Characteristics: - Does not increase reference count - Automatically becomes `nil` - Must always be optional

Used when: - Parent owns child - Child references parent

Unowned References

```
unowned let owner: Owner
```

Characteristics: - Does not increase reference count - Never becomes `nil` - Causes a crash if accessed after deallocation

Use `unowned` **only when the lifetime of the referenced object is guaranteed**.

Retain Cycles (Most Important Topic)

Classic Retain Cycle Example

```
class A {
    var b: B?
```

```
}

class B {
    var a: A?
}
```

Both objects hold strong references to each other. Their reference counts never reach zero → **memory leak**.

Correct Fix

```
class B {
    weak var a: A?
}
```

Real-World Retain Cycle (Very Common)

```
viewModel.onUpdate = {
    self.updateUI()
}
```

Here: - Closure captures `self` strongly - ViewController never deallocates

Correct version:

```
viewModel.onUpdate = { [weak self] in
    self?.updateUI()
}
```

ARC and Value Types

ARC applies **only to reference types**.

Type	Managed by ARC
class	Yes
struct	No
enum	No

Value types use **copy semantics**, not reference counting.

Autorelease Pool

`autoreleasepool` is used to control the lifetime of temporary objects, especially when working with Objective-C APIs.

```
autoreleasepool {
    // temporary objects
}
```

Common use cases: - Image processing loops - Background batch operations - Legacy Objective-C code

---

Common Mistakes

- “ARC automatically removes all memory leaks”
- “Weak and unowned are the same”
- “Swift uses garbage collection”
- “Structs are managed by ARC”

These answers immediately reduce interview confidence.

Follow-Up Questions Interviewers Ask

- What is the difference between `weak` and `unowned`?
- When is `deinit` called?
- How do you debug memory leaks?
- What Instruments tool do you use to find retain cycles?
- What is an autorelease pool?

Key Takeaways

- ARC simplifies memory management but does not eliminate bugs
- Retain cycles are the most common real-world memory issue
- Understanding object lifetimes is a senior-level skill
- Memory management mistakes lead to crashes and leaks

This Preview Ends Here

You have reached the end of the sample.

The full book continues with value vs reference types, Copy-on-Write, struct vs class vs actor, protocol-oriented programming, generics, Swift Concurrency, actors, GCD, dependency injection, networking, caching, UIKit vs SwiftUI, SwiftUI rendering, the RunLoop, and a closing chapter on debugging and interview strategy.

If this sample helped you see how the answers are structured — the question, why it is asked, the 2–3 minute version, then the traps — the rest of the book is the same format, applied to the rest of the platform round.
