SWIFT INTERVIEW Q&A — DESCRIPTIVE + CODE EXAMPLES

Explain the concept, then SHOW it with code

The exact format for a technical interview

Q1: What is the difference between a struct and a class?

Structs are **value types** — copied on assignment. Classes are **reference types** — shared via reference. This fundamental difference affects memory, thread safety, and when you use each.

```swift
// STRUCT — Value Type (copied)
struct Rate {
    var value: Decimal
    let currency: String
}

var rate1 = Rate(value: 1.25, currency: "EUR")
var rate2 = rate1       // COPY — rate2 is independent
rate2.value = 1.30
// rate1.value is still 1.25 — not affected

// CLASS — Reference Type (shared)
class RateService {
    var currentRate: Decimal = 1.25
}

let service1 = RateService()
let service2 = service1  // SAME object in memory
service2.currentRate = 1.30
// service1.currentRate is also 1.30 — both point to same object
```

Key differences in practice:

- **Memory:** Structs live on the stack (fast), classes on the heap (needs ARC)
- **Thread safety:** Structs are inherently safe (each thread gets its own copy). Classes need locks or actors
- **Inheritance:** Only classes support it. Structs use protocols instead
- **Identity:** Classes have identity (`===` checks same object). Structs only have equality (`==`)
- **Mutability:** `let` on a struct = everything immutable. `let` on a class = can't reassign reference, but CAN change properties

```swift
let account = RateService()
account.currentRate = 5.0  // ✅ OK — changing property, not reference
// account = RateService()  // ❌ Error — can't reassign let reference
```

When to use which in fintech:

- `struct` → Rate, Transaction, Money, Quote (pure data, should copy safely)
- `class` → TransferService, ViewModel, NetworkManager (shared state, identity matters)
- `actor` → RateCache, BalanceStore (thread-safe shared state)

Q2: What are optionals? How do you safely unwrap them?

Optionals represent a value that might or might not exist — it's Swift's way of forcing you to handle the absence of data at compile time instead of crashing at runtime with null pointer errors.

```swift
```

```swift
var exchangeRate: Decimal? = nil  // No rate fetched yet

// 1. guard let — early return (BEST for functions)
func processRate(_ rate: Decimal?) {
    guard let rate = rate else {
        print("No rate available")
        return  // Exit early
    }
    // rate is non-optional from here — clean, un-nested
    print("Processing \(rate)")
}

// 2. if let — when nil case doesn't need early return
if let rate = exchangeRate {
    print("Rate: \(rate)")
} else {
    print("No rate")
}

// 3. Nil coalescing (??) — provide default
let display = exchangeRate ?? Decimal(0)  // Use 0 if nil

// 4. Optional chaining (?.) — safe property access
struct User { var wallet: Wallet? }
struct Wallet { var balance: Decimal? }
let user = User(wallet: Wallet(balance: 500))
let balance = user.wallet?.balance  // Optional(500)
// If wallet is nil, whole chain returns nil — no crash

// 5. map on optional — transform if exists
let doubled = exchangeRate.map { $0 * 2 }  // nil if rate is nil, Optional(2.50) if 1.25

// 6. Multiple optional binding in one guard
func transfer(amount: Decimal?, recipient: String?, rate: Decimal?) {
    guard let amount = amount,
          let recipient = recipient,
          let rate = rate,
          amount > 0 else { return }
    print("Send \(amount) to \(recipient) at \(rate)")
}
```

The golden rule: Never force unwrap (`!`) in production. The only exception is known-valid literals: `URL(string: "https://api.wise.com")!`

Q3: How does ARC work? What are strong, weak, and unowned?

ARC (Automatic Reference Counting) manages memory for class instances. Every strong reference increments a counter; when it reaches zero, the object is deallocated.

```swift
class TransferViewModel {
    var transfers: [String] = []
    deinit { print("ViewModel deallocated") }
}

var vm1: TransferViewModel? = TransferViewModel()  // Reference count: 1
var vm2 = vm1                                      // Reference count: 2
vm1 = nil                                          // Reference count: 1 (still alive)
vm2 = nil                                          // Reference count: 0 → deallocated ✅
```

The problem — Retain Cycles:

Two objects holding strong references to each other → neither ever reaches zero → memory leak.

```swift
// ❌ RETAIN CYCLE
class Parent {
    var child: Child?
    deinit { print("Parent freed") }  // Never called!
}
class Child {
    var parent: Parent?   // Strong reference back → CYCLE
    deinit { print("Child freed") }   // Never called!
}

var p: Parent? = Parent()
var c: Child? = Child()
p?.child = c    // Parent → Child (strong)
c?.parent = p   // Child → Parent (strong) → CYCLE!
p = nil; c = nil  // Neither deallocated — LEAK

// ✅ FIX: Make one reference weak
class ChildFixed {
    weak var parent: Parent?  // Weak — doesn't increase count
    deinit { print("Child freed") }
}
```

When to use each:

```swift
// STRONG (default) — ownership. Parent owns child.
class ViewController {
    let tableView = UITableView()  // Strong — VC owns the table view
}

// WEAK — delegate pattern, closures. Must be var + Optional.
protocol TransferDelegate: AnyObject {}
class TransferManager {
    weak var delegate: TransferDelegate?  // Always weak for delegates
}

// UNOWNED — when you're CERTAIN the ref outlives the owner.
class CreditCard {
    unowned let owner: Customer  // Card can't exist without owner
    init(owner: Customer) { self.owner = owner }
}
// ⚠️ Crashes if accessed after owner is deallocated!
```

Q4: Explain closures and [weak self]. Why does it matter?

Closures are self-contained blocks of code you can pass around. They "capture" variables from surrounding scope — which is powerful but can cause memory leaks when capturing `self` in a class.

```swift
// Basic closure syntax
let multiply: (Int, Int) -> Int = { a, b in a * b }
multiply(3, 4)  // 12

// Trailing closure with shorthand
let sorted = [3, 1, 4].sorted { $0 < $1 }  // [1, 3, 4]
```

@escaping vs non-escaping:

```swift
```

```swift
class NetworkManager {
    // Non-escaping (default) — called within function's lifetime
    func transform(data: Data, using: (Data) -> String) -> String {
        return using(data)  // Called immediately, then discarded
    }

    // @escaping — stored or called AFTER function returns
    func fetch(completion: @escaping (Data) -> Void) {
        DispatchQueue.global().async {
            let data = Data()
            completion(data)  // Called later — must be @escaping
        }
    }
}
```

[weak self] — preventing retain cycles in closures:

```swift
class RateViewModel {
    var rate: Decimal = 0
    var onUpdate: (() -> Void)?

    func startPolling() {
        // ❌ RETAIN CYCLE: self → onUpdate → self
        // onUpdate = { self.rate = 1.25 }

        // ✅ [weak self] breaks the cycle
        onUpdate = { [weak self] in
            guard let self = self else { return }  // Exit if deallocated
            self.rate = 1.25
        }
    }

    deinit { print("ViewModel freed") }
}

// With weak self: ViewModel deallocates normally ✅
// Without weak self: ViewModel NEVER deallocates — memory leak ❌
```

The rule: Use `[weak self]` in any escaping closure on a class instance — network callbacks, timers, Combine subscriptions, notification observers.

Q5: What is Protocol-Oriented Programming?

POP means defining behavior through protocols and composing capabilities, instead of building deep class inheritance hierarchies. Swift favors this because protocols work with structs, enable multiple conformance, and produce more modular code.

```swift
// Instead of a base class hierarchy:
// PaymentService → CardPaymentService → VisaPaymentService → ...

// Define small, focused protocols:
protocol Payable {
    func processPayment(amount: Decimal) throws -> String
}

protocol Refundable {
    func processRefund(transactionId: String) throws
}

protocol Recurring {
    func schedule(interval: TimeInterval)
}

// Default implementation via extension:
extension Payable {
    func processPayment(amount: Decimal) throws -> String {
        guard amount > 0 else { throw PaymentError.invalidAmount }
        return UUID().uuidString
    }
}

// Compose capabilities — each type picks what it needs:
struct BankTransfer: Payable, Recurring {
    func schedule(interval: TimeInterval) { /* ... */ }
    // Gets processPayment() for free from default implementation
}

struct CardPayment: Payable, Refundable {
    func processRefund(transactionId: String) { /* ... */ }
}

// A struct can't inherit from a class, but it CAN conform to protocols.
// This is why POP is essential — Swift's primary types are structs.
```

The killer use case — Dependency Injection for testability:

swift

```swift
protocol NetworkSession {
    func data(from url: URL) async throws -> (Data, URLResponse)
}

// Real implementation — production:
extension URLSession: NetworkSession {}

// Mock implementation — testing:
class MockSession: NetworkSession {
    var mockData: Data = Data()
    var mockStatusCode: Int = 200

    func data(from url: URL) async throws -> (Data, URLResponse) {
        let response = HTTPURLResponse(url: url, statusCode: mockStatusCode,
                                       httpVersion: nil, headerFields: nil)!
        return (mockData, response)
    }
}

// The service depends on the PROTOCOL, not concrete URLSession:
class RateClient {
    private let session: NetworkSession
    init(session: NetworkSession = URLSession.shared) {
        self.session = session
    }

    func fetchRate(from: String, to: String) async throws -> Decimal {
        let url = URL(string: "https://api.wise.com/v1/rates?source=\(from)&target=\(to)")!
        let (data, _) = try await session.data(from: url)
        return try JSONDecoder().decode(RateResponse.self, from: data).rate
    }
}

// Production: RateClient()              — uses real URLSession
// Test:       RateClient(session: mock)  — uses mock with fake data
```

Q6: Explain generics. What are `some` and `any`?

Generics let you write flexible, reusable code that works with any type while keeping compile-time type safety. Instead of writing separate functions for each type, you write it once with a type parameter.

swift

```swift
// Without generics — need separate functions:
func findDuplicateInts(_ array: [Int]) -> Set<Int> { /* ... */ }
func findDuplicateStrings(_ array: [String]) -> Set<String> { /* ... */ }

// With generics — one function for any Hashable type:
func findDuplicates<T: Hashable>(in array: [T]) -> Set<T> {
    var seen: Set<T> = []
    var dupes: Set<T> = []
    for item in array {
        if !seen.insert(item).inserted { dupes.insert(item) }
    }
    return dupes
}

findDuplicates(in: [1, 2, 3, 2, 4, 3])      // {2, 3}
findDuplicates(in: ["USD", "EUR", "USD"])    // {"USD"}

// Generic type with constraint:
struct Cache<Key: Hashable, Value> {
    private var storage: [Key: Value] = [:]

    mutating func set(_ key: Key, _ value: Value) { storage[key] = value }
    func get(_ key: Key) -> Value? { storage[key] }
}

var rateCache = Cache<String, Decimal>()
rateCache.set("USD-EUR", 0.92)
```

`some` vs `any` — opaque vs existential types:

```swift
```

```swift
protocol Shape {
    func area() -> Double
}
struct Circle: Shape { let radius: Double; func area() -> Double { .pi * radius * radius } }
struct Square: Shape { let side: Double; func area() -> Double { side * side } }

// `some Shape` — compiler knows the EXACT type, but caller doesn't see it
// Always returns the SAME concrete type. Enables static dispatch (faster).
func makeShape() -> some Shape {
    Circle(radius: 5)  // Always a Circle — compiler can optimize
}

// `any Shape` — can be DIFFERENT concrete types at runtime
// Requires boxing and dynamic dispatch (slower).
var shapes: [any Shape] = [Circle(radius: 3), Square(side: 4)]

// RULE OF THUMB:
// Use `some` when returning a single type (function returns, SwiftUI body)
// Use `any` when storing mixed types (array of different shapes)
```

Q7: Explain async/await and structured concurrency.

Async/await lets you write asynchronous code that reads like synchronous code. An `async` function can suspend without blocking the thread, and `await` marks where that suspension might happen.

```swift
```

```
// BEFORE async/await — callback hell:
func fetchRate(completion: @escaping (Result<Decimal, Error>) -> Void) {
    URLSession.shared.dataTask(with: url) { data, response, error in
        if let error = error { completion(.failure(error)); return }
        guard let data = data else { return }
        do {
            let rate = try JSONDecoder().decode(RateResponse.self, from: data)
            completion(.success(rate.value))
        } catch {
            completion(.failure(error))
        }
    }.resume()
}

// AFTER async/await — clean, readable:
func fetchRate(from: String, to: String) async throws -> Decimal {
    let url = URL(string: "https://api.wise.com/rates?source=\(from)&target=\(to)")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(RateResponse.self, from: data).rate
}
```

Parallel execution with `async let`:

```
swift

// Run multiple operations concurrently:
func loadDashboard() async throws -> Dashboard {
    async let balance = fetchBalance()      // Starts immediately
    async let transfers = fetchTransfers()  // Starts immediately
    async let rate = fetchRate(from: "USD", to: "EUR")  // Starts immediately

    // All three run in parallel, await ALL results:
    return try await Dashboard(
        balance: balance,
        transfers: transfers,
        rate: rate
    )
}
// Without async let: sequential = 3 seconds (1s + 1s + 1s)
// With async let: parallel = 1 second (all run simultaneously)
```

TaskGroup for dynamic parallelism:

```
swift
```

```
// When you don't know the number of tasks at compile time:
func fetchRatesForAllPairs(_ pairs: [(String, String)]) async throws -> [String: Decimal] {
    try await withThrowingTaskGroup(of: (String, Decimal).self) { group in
        for (from, to) in pairs {
            group.addTask {
                let rate = try await fetchRate(from: from, to: to)
                return ("\(from)-\(to)", rate)
            }
        }
        var rates: [String: Decimal] = [:]
        for try await (pair, rate) in group {
            rates[pair] = rate
        }
        return rates
    }
}
```

Structured concurrency means child tasks are scoped within their parent. When the parent completes or is cancelled, all children are automatically cancelled — no orphaned background work.

Q8: What is an Actor? How does it prevent data races?

An actor is a reference type that protects its mutable state from concurrent access. The compiler enforces that only one task can access an actor's state at a time, preventing data races by design.

swift

```
// ❌ CLASS — NOT thread-safe:
class UnsafeRateCache {
    private var cache: [String: Decimal] = [:]

    func set(_ pair: String, rate: Decimal) {
        cache[pair] = rate  // Two threads doing this simultaneously = DATA RACE
    }

    func get(_ pair: String) -> Decimal? {
        cache[pair]  // Reading while another thread writes = CRASH or garbage
    }
}

// ✅ ACTOR — thread-safe by compiler enforcement:
actor SafeRateCache {
    private var cache: [String: (rate: Decimal, expiry: Date)] = [:]

    func set(_ pair: String, rate: Decimal, ttl: TimeInterval = 30) {
        cache[pair] = (rate, Date().addingTimeInterval(ttl))
    }

    func get(_ pair: String) -> Decimal? {
        guard let entry = cache[pair], entry.expiry > Date() else { return nil }
        return entry.rate
    }

    func clearExpired() {
        cache = cache.filter { $0.value.expiry > Date() }
    }
}

// All access requires `await` — compiler forces you to acknowledge potential waiting:
let cache = SafeRateCache()
await cache.set("USD-EUR", rate: 0.92)
let rate = await cache.get("USD-EUR")
```

@MainActor — for UI thread safety:

```
swift
```

```swift
@MainActor
class TransferViewModel: ObservableObject {
    @Published var transfers: [Transfer] = []
    @Published var isLoading = false

    func load() async {
        isLoading = true  // Guaranteed main thread by @MainActor
        let data = await api.fetchTransfers()  // Can run on any thread
        transfers = data  // Back on main thread automatically
        isLoading = false
    }
}
```

Sendable marks types safe to pass between actors:

```swift
struct TransferData: Sendable { // Struct = implicitly Sendable
    let id: String
    let amount: Decimal
}

final class Config: Sendable {  // Class must be final + all let properties
    let apiURL: String
    init(apiURL: String) { self.apiURL = apiURL }
}
```

Q9: Explain MVVM pattern.

MVVM separates the app into Model (data + rules), View (UI), and ViewModel (mediator that transforms Model data for the View). The View observes the ViewModel and updates automatically when state changes.

```swift
```

```swift
// MODEL — pure data + business logic
struct TransferQuote {
    let sourceAmount: Decimal
    let fee: Decimal
    let rate: Decimal
    let targetCurrency: String

    var targetAmount: Decimal { (sourceAmount - fee) * rate }
    var isExpired: Bool { Date() > expiresAt }
    let expiresAt: Date
}

// VIEWMODEL — transforms Model for display, handles user actions
// Does NOT import SwiftUI — fully testable with XCTest
class TransferViewModel: ObservableObject {
    @Published var amountText: String = ""
    @Published var quote: TransferQuote?
    @Published var isLoading = false
    @Published var error: String?
    @Published var formattedRate: String = ""
    @Published var formattedFee: String = ""

    private let rateService: RateServiceProtocol  // Protocol — injectable

    init(rateService: RateServiceProtocol = RateService()) {
        self.rateService = rateService
    }

    func fetchQuote() async {
        guard let amount = Decimal(string: amountText), amount > 0 else {
            error = "Enter a valid amount"
            return
        }
        isLoading = true
        defer { isLoading = false }

        do {
            let quote = try await rateService.getQuote(amount: amount, to: "EUR")
            self.quote = quote
            formattedRate = "1 USD = \(quote.rate) EUR"
            formattedFee = "Fee: $\(quote.fee)"
            error = nil
        } catch {
            self.error = "Failed to fetch rate"
        }
    }
```

```swift
}

// VIEW — SwiftUI, just renders ViewModel state
struct TransferView: View {
    @StateObject private var vm = TransferViewModel()

    var body: some View {
        VStack {
            TextField("Amount", text: $vm.amountText)

            if vm.isLoading { ProgressView() }

            if let error = vm.error { Text(error).foregroundColor(.red) }

            if let _ = vm.quote {
                Text(vm.formattedRate)
                Text(vm.formattedFee)
                Button("Send") { /* confirm */ }
            }
        }
        .task { await vm.fetchQuote() }
    }
}
```

Why MVVM works for fintech:

- ViewModel has no UIKit/SwiftUI imports → testable independently
- Fee calculations, validation, rate formatting all testable with XCTest
- Same ViewModel can power both SwiftUI and UIKit views during migration

---

Q10: What is Dependency Injection? Why is it critical?

DI means an object receives its dependencies from outside rather than creating them internally. This makes code testable, modular, and loosely coupled.

```swift
```

```
// ❌ WITHOUT DI — tightly coupled, untestable:
class BadTransferService {
    private let client = URLSession.shared  // Hardcoded — can't swap for testing

    func sendMoney(amount: Decimal) async throws -> String {
        let (data, _) = try await client.data(from: transferURL)
        return try JSONDecoder().decode(TransferResponse.self, from: data).id
    }
}
// How do you test this without making real network calls? You can't.

// ✅ WITH DI — protocol-based, testable:
protocol NetworkSession {
    func data(from url: URL) async throws -> (Data, URLResponse)
}

extension URLSession: NetworkSession {}  // Production implementation

class MockSession: NetworkSession {      // Test implementation
    var mockData: Data = Data()
    var shouldFail = false

    func data(from url: URL) async throws -> (Data, URLResponse) {
        if shouldFail { throw URLError(.badServerResponse) }
        return (mockData, HTTPURLResponse(url: url, statusCode: 200,
                                          httpVersion: nil, headerFields: nil)!)
    }
}

class GoodTransferService {
    private let session: NetworkSession

    init(session: NetworkSession = URLSession.shared) {  // Default for production
        self.session = session
    }

    func sendMoney(amount: Decimal) async throws -> String {
        let (data, _) = try await session.data(from: transferURL)
        return try JSONDecoder().decode(TransferResponse.self, from: data).id
    }
}

// PRODUCTION — just works with default:
let service = GoodTransferService()

// TESTING — inject mock:
```

```
let mock = MockSession()
mock.mockData = """{"id": "test-123"}""".data(using: .utf8)!
let testService = GoodTransferService(session: mock)
let id = try await testService.sendMoney(amount: 100)
assert(id == "test-123")  // ✅ No network call, runs in milliseconds
```

Q11: Explain Codable. How do you handle API JSON?

Codable is Swift's protocol for converting between Swift types and external representations like JSON. When properties match JSON keys, the compiler generates code automatically.

swift

```
// SIMPLE — automatic when names match:
struct Transfer: Codable {
    let id: String
    let amount: Decimal
    let status: String
}

let json = """
{"id": "abc-123", "amount": 1000.50, "status": "completed"}
""".data(using: .utf8)!

let transfer = try JSONDecoder().decode(Transfer.self, from: json)

// CUSTOM KEYS — when API uses snake_case:
struct WiseRate: Codable {
    let sourceCurrency: String
    let targetCurrency: String
    let midMarketRate: Decimal

    enum CodingKeys: String, CodingKey {
        case sourceCurrency = "source_currency"
        case targetCurrency = "target_currency"
        case midMarketRate = "mid_market_rate"
    }
}

// OR use keyDecodingStrategy (simpler for consistent snake_case):
let decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase
let rate = try decoder.decode(WiseRate.self, from: jsonData)

// NESTED JSON:
struct APIResponse<T: Codable>: Codable {
    let status: String
    let data: T
    let pagination: Pagination?

    struct Pagination: Codable {
        let nextCursor: String?
        let hasMore: Bool
    }
}

let response = try decoder.decode(APIResponse<[Transfer]>.self, from: json)
let transfers = response.data  // [Transfer]
```

```swift
// CUSTOM DECODING — when API is inconsistent:
struct FlexibleRate: Codable {
    let rate: Decimal

    init(from decoder: Decoder) throws {
        let container = try decoder.singleValueContainer()
        // API sometimes sends number, sometimes string
        if let decimal = try? container.decode(Decimal.self) {
            rate = decimal
        } else if let string = try? container.decode(String.self),
                  let decimal = Decimal(string: string) {
            rate = decimal
        } else {
            throw DecodingError.dataCorrupted(
                .init(codingPath: decoder.codingPath,
                      debugDescription: "Expected Decimal or String"))
        }
    }
}
```

Fintech rule: Always decode monetary amounts as `Decimal`, never `Double`. `0.1 + 0.2` in Double is `0.30000000000000004`, not `0.3`.

Q12: Explain SwiftUI state management property wrappers.

SwiftUI is declarative — UI is a function of state. Property wrappers control where state lives and who owns it.

swift

```swift
// @State — local state owned by the View
struct TransferFormView: View {
    @State private var amount = ""        // Simple local state
    @State private var showConfirm = false

    var body: some View {
        TextField("Amount", text: $amount)  // $ creates a Binding
        Button("Send") { showConfirm = true }
            .sheet(isPresented: $showConfirm) { ConfirmView() }
    }
}

// @Binding — child modifies parent's state
struct AmountField: View {
    @Binding var amount: String  // Doesn't own it — parent does

    var body: some View {
        TextField("Amount", text: $amount)
    }
}

// @StateObject — creates and OWNS an ObservableObject (created ONCE)
struct TransferScreen: View {
    @StateObject private var vm = TransferVM()  // Created once, survives re-renders

    var body: some View {
        TransferDetailView(vm: vm)
    }
}

// @ObservedObject — observes but DOESN'T own (passed from parent)
struct TransferDetailView: View {
    @ObservedObject var vm: TransferVM  // Passed in, not created here

    var body: some View {
        Text(vm.formattedRate)
    }
}

// @Published — inside ViewModel, triggers UI updates on change
class TransferVM: ObservableObject {
    @Published var amount = ""
    @Published var rate: Decimal?       // View re-renders when these change
    @Published var isLoading = false
    @Published var error: String?
```

```swift
    func fetchRate() async {
        isLoading = true
        defer { isLoading = false }
        rate = try? await rateService.getRate()
    }
}

// @EnvironmentObject — shared across view hierarchy without passing
class UserSession: ObservableObject {
    @Published var isAuthenticated = false
    @Published var userId: String?
}

// Inject at top:
// ContentView().environmentObject(UserSession())
// Access anywhere below:
struct ProfileView: View {
    @EnvironmentObject var session: UserSession
    var body: some View { Text(session.userId ?? "Not logged in") }
}
```

Quick decision guide:

I need to...	Use
Store simple local UI state	`@State`
Let child modify parent's state	`@Binding`
Create a ViewModel for this screen	`@StateObject`
Receive a ViewModel from parent	`@ObservedObject`
Share state across many screens	`@EnvironmentObject`
Trigger UI updates from ViewModel	`@Published`

Q13: How do higher-order functions work? map, filter, reduce, etc.

Higher-order functions take or return other functions. Swift's collection methods let you transform data declaratively without manual loops.

```swift
```

```
struct Transaction {
    let amount: Decimal
    let currency: String
    let type: String  // "send", "receive", "card"
}

let transactions = [
    Transaction(amount: 100, currency: "USD", type: "send"),
    Transaction(amount: 50,  currency: "EUR", type: "receive"),
    Transaction(amount: 200, currency: "USD", type: "send"),
    Transaction(amount: 75,  currency: "GBP", type: "card"),
    Transaction(amount: 30,  currency: "EUR", type: "send"),
]

// MAP — transform each element
let amounts = transactions.map { $0.amount }
// [100, 50, 200, 75, 30]

// FILTER — keep elements matching condition
let sends = transactions.filter { $0.type == "send" }
// 3 transactions where type == "send"

// REDUCE — combine all elements into single value
let total = transactions.reduce(Decimal(0)) { $0 + $1.amount }
// 455

// COMPACTMAP — transform + remove nils
let strings = ["100", "abc", "200"]
let numbers = strings.compactMap { Decimal(string: $0) }
// [100, 200] — "abc" became nil and was removed

// FLATMAP — transform + flatten nested arrays
let grouped = [["USD", "EUR"], ["GBP", "SGD"]]
let all = grouped.flatMap { $0 }
// ["USD", "EUR", "GBP", "SGD"]

// CHAINING — combine multiple operations
let usdSendTotal = transactions
    .filter { $0.currency == "USD" && $0.type == "send" }
    .map { $0.amount }
    .reduce(Decimal(0), +)
// 300 (100 + 200)

// DICTIONARY(grouping:) — group by key
let byCurrency = Dictionary(grouping: transactions) { $0.currency }
// ["USD": [txn1, txn3], "EUR": [txn2, txn5], "GBP": [txn4]]
```

```swift
// SORTED with multiple criteria
let sorted = transactions.sorted {
    if $0.currency != $1.currency { return $0.currency < $1.currency }
    return $0.amount > $1.amount  // Same currency → by amount desc
}

// FIRST(where:) — find first match
let firstBig = transactions.first { $0.amount > 100 }

// CONTAINS(where:)
let hasGBP = transactions.contains { $0.currency == "GBP" }  // true

// ALLSATISFY — check ALL match
let allSends = transactions.allSatisfy { $0.type == "send" }  // false
```

Q14: Decimal vs Double — why does it matter for fintech?

Double uses binary floating-point which cannot exactly represent many decimal fractions. For money, this is unacceptable.

```swift
```

```swift
// ❌ DOUBLE — binary floating-point:
let a: Double = 0.1
let b: Double = 0.2
print(a + b)          // 0.30000000000000004  ← NOT 0.3!
print(a + b == 0.3)   // false!

// For a $1000 transfer at 0.1% fee:
let feeDouble: Double = 1000.0 * 0.001
print(feeDouble)      // 1.0 — OK here, but compound operations accumulate error

// ✅ DECIMAL — base-10 representation:
let x: Decimal = Decimal(string: "0.1")!
let y: Decimal = Decimal(string: "0.2")!
print(x + y)          // 0.3  ← Exact!
print(x + y == Decimal(string: "0.3")!)  // true ✅

// IMPORTANT: Create Decimal from STRING, not from Double literal:
let good = Decimal(string: "0.1")!  // ✅ Exact 0.1
let bad = Decimal(0.1)              // ❌ Converts imprecise Double to Decimal

// In Codable:
struct Rate: Codable {
    let amount: Decimal  // NOT Double
    let fee: Decimal     // NOT Double
    let rate: Decimal    // NOT Double
}

// Formatting for display:
let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.currencyCode = "USD"
formatter.string(from: 1234.56 as NSDecimalNumber)  // "$1,234.56"
```

Interview tip: Mentioning Decimal vs Double proactively when discussing any financial calculation shows you understand fintech fundamentals. This is one of the simplest ways to demonstrate domain expertise.

Q15: How do you write testable code? What's your testing strategy?

Testability is an architectural decision, not an afterthought. The three pillars are: protocol-based DI, pure functions, and separation of concerns.

swift

```
// PROTOCOL-BASED DI — inject mocks for testing:
protocol RateServiceProtocol {
    func getRate(from: String, to: String) async throws -> Decimal
}

class MockRateService: RateServiceProtocol {
    var mockRate: Decimal = 0.92
    var shouldFail = false

    func getRate(from: String, to: String) async throws -> Decimal {
        if shouldFail { throw TransferError.networkError }
        return mockRate
    }
}

// ViewModel depends on protocol, not concrete type:
class TransferVM2: ObservableObject {
    private let rateService: RateServiceProtocol
    @Published var rate: Decimal?
    @Published var error: String?

    init(rateService: RateServiceProtocol = RealRateService()) {
        self.rateService = rateService
    }

    func fetchRate() async {
        do {
            rate = try await rateService.getRate(from: "USD", to: "EUR")
        } catch {
            self.error = "Failed to fetch rate"
        }
    }
}

// UNIT TESTS:

func testFetchRate_success() async {
    let mock = MockRateService()
    mock.mockRate = 0.92
    let vm = TransferVM2(rateService: mock)

    await vm.fetchRate()

    XCTAssertEqual(vm.rate, 0.92)
    XCTAssertNil(vm.error)
}
```

```
func testFetchRate_failure() async {
    let mock = MockRateService()
    mock.shouldFail = true
    let vm = TransferVM2(rateService: mock)

    await vm.fetchRate()

    XCTAssertNil(vm.rate)
    XCTAssertEqual(vm.error, "Failed to fetch rate")
}

// PURE FUNCTIONS — always return same output for same input:
func calculateFee(amount: Decimal, percentage: Decimal, minimum: Decimal) -> Decimal {
    max(amount * percentage, minimum)
}

func testFeeCalculation() {
    XCTAssertEqual(calculateFee(amount: 1000, percentage: 0.005, minimum: 1), 5)
    XCTAssertEqual(calculateFee(amount: 100, percentage: 0.005, minimum: 1), 1) // hits minimum
    XCTAssertEqual(calculateFee(amount: 0, percentage: 0.005, minimum: 1), 1)   // zero amount
}

// INJECTABLE TIME — for testing time-dependent logic:
protocol TimeProvider {
    func now() -> Date
}

class MockTime: TimeProvider {
    var current = Date()
    func now() -> Date { current }
    func advance(by seconds: TimeInterval) {
        current = current.addingTimeInterval(seconds)
    }
}

// Cache with injectable time:
class RateCache2 {
    private let time: TimeProvider
    private var cache: [String: (rate: Decimal, expiry: Date)] = [:]

    init(time: TimeProvider = SystemTime()) {
        self.time = time
    }

    func get(_ pair: String) -> Decimal? {
        guard let entry = cache[pair], entry.expiry > time.now() else { return nil }
```

```
        return entry.rate
    }
}

// Test:
func testCache_expiresAfterTTL() {
    let mockTime = MockTime()
    let cache = RateCache2(time: mockTime)
    cache.set("USD-EUR", rate: 0.92, ttl: 30)

    XCTAssertEqual(cache.get("USD-EUR"), 0.92)  // ✅ Fresh

    mockTime.advance(by: 31)  // Fast-forward 31 seconds

    XCTAssertNil(cache.get("USD-EUR"))  // ✅ Expired — no real waiting!
}
```

Testing pyramid: Unit tests (70%) → Integration tests (20%) → UI tests (10%). Most bugs are caught cheaply at the unit level.

Q16: What is certificate pinning and why is it essential for fintech?

Certificate pinning hardcodes the server's expected certificate or public key in your app, rejecting connections where the certificate doesn't match — even if it's signed by a trusted authority. This prevents man-in-the-middle attacks.

swift

```swift
// Without pinning: attacker with a fake (but CA-signed) certificate
// can intercept traffic between app and Wise's servers.
// With pinning: app rejects the fake certificate.

class PinnedSessionDelegate: NSObject, URLSessionDelegate {
    // The expected public key hash (from Wise's certificate)
    private let pinnedPublicKeyHash = "sha256/AAAAAAA...="

    func urlSession(_ session: URLSession,
                    didReceive challenge: URLAuthenticationChallenge,
                    completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {

        guard let serverTrust = challenge.protectionSpace.serverTrust,
              let certificate = SecTrustGetCertificateAtIndex(serverTrust, 0) else {
            completionHandler(.cancelAuthenticationChallenge, nil)
            return
        }

        // Extract server's public key
        let serverPublicKey = SecCertificateCopyKey(certificate)
        let serverKeyHash = hash(of: serverPublicKey)

        // Compare with pinned key
        if serverKeyHash == pinnedPublicKeyHash {
            completionHandler(.useCredential, URLCredential(trust: serverTrust))
        } else {
            // Key doesn't match — possible MITM attack!
            completionHandler(.cancelAuthenticationChallenge, nil)
        }
    }
}

// Create session with pinning delegate:
let session = URLSession(
    configuration: .default,
    delegate: PinnedSessionDelegate(),
    delegateQueue: nil
)
```

Why this matters at Wise: Wise processes £36 billion per quarter. A compromised connection could let an attacker redirect transfers, steal credentials, or modify transaction amounts. Certificate pinning is non-negotiable for any fintech app.

Public key pinning vs certificate pinning: I'd recommend public key pinning because the public key survives certificate renewals (typically yearly), so the app doesn't break when Wise renews their certificate. Certificate pinning would require an app update each time.