Swift Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a thread-safe singleton pattern in Swift

Solution:

Here's a thread-safe singleton implementation using Swift:

```
class DataManager {
    static let shared = DataManager()
    private var data: [String: Any] = [:]
    private let queue = DispatchQueue(label: "com.datamanager.queue")
    private init() {}

    func setValue(_ value: Any, forKey key: String) {
        queue.async { self.data[key] = value }
    }
}
```

Key points:

- Uses static let for thread-safe initialization
- Private initializer prevents external instantiation
- DispatchQueue ensures thread-safe operations

2. How would you implement a custom memory management solution for a cache system?

Implementation:

```
final class Cache {
    private var storage: [Key: Value] = [:]
    private let maxSize: Int
    private let queue = DispatchQueue(label: "cache.queue")

    func set(_ value: Value, for key: Key) {
        queue.async {
            if self.storage.count >= self.maxSize { self.storage.removeValue(forKey: self.storage.keys.first!) }
            self.storage[key] = value
        }
    }}
```

Features:

- Generic implementation
- Thread-safe operations
- Automatic cleanup when reaching maxSize

3. Create a protocol that implements the Observer pattern with generics

Implementation:

```
protocol Observable {
    associatedtype DataType
    var observers: [(DataType) -> Void] { get set }
    func subscribe(_ handler: @escaping (DataType) -> Void)
    func notify(_ data: DataType)
}
```

```
extension Observable {
    mutating func subscribe(_ handler: @escaping (DataType) -> Void) {
        observers.append(handler)
    }}
```

Benefits:

- Type-safe implementation
- Flexible data types
- Decoupled communication

4. Implement a custom Result Builder for a DSL

Solution:

```
@resultBuilder
struct ArrayBuilder {
    static func buildBlock(_ components: [T]...) -> [T] {
        components.flatMap { $0 }
    }
    static func buildExpression(_ expression: T) -> [T] {
        [expression]
    }
    static func buildOptional(_ component: [T]?) -> [T] {
        component ?? []
    }}
```

Key features:

- Generic implementation
- Supports optional components
- Composable syntax

5. Design a type-safe networking layer using protocols and generics

Implementation:

```
protocol Endpoint {
    associatedtype Response: Decodable
    var path: String { get }
    var method: HTTPMethod { get }
}

struct NetworkManager {
    func request(_ endpoint: E) async throws -> E.Response {
        // Implementation
    }}
```

Advantages:

- Type-safe responses
- Protocol-oriented design
- Async/await support

6. Implement a custom property wrapper for data persistence

Solution:

```
@propertyWrapper
struct Persisted {
    private let key: String
    var wrappedValue: T {
        get { UserDefaults.standard.object(forKey: key) as? T ?? defaultValue }
        set { UserDefaults.standard.set(newValue, forKey: key) }
    }
    private let defaultValue: T
```

Features:

- Generic implementation
- Automatic persistence
- Default value support

7. Create a custom async sequence for pagination

Implementation:

```
struct PaginatedSequence: AsyncSequence {
    typealias Element = [Item]
    let pageSize: Int

    struct AsyncIterator: AsyncIteratorProtocol {
        var currentPage = 0
        mutating func next() async throws -> [Item]? {
            // Fetch logic
        }}
```

Benefits:

- Native async/await support
- Efficient memory usage
- Lazy loading

8. Implement a type-safe dependency injection container

Solution:

```
final class Container {
    private var services: [String: Any] = [:]

    func register(_ type: T.Type, factory: @escaping () -> T) {
        services[String(describing: type)] = factory
    }

    func resolve() -> T? {
        (services[String(describing: T.self)] as? () -> T)?()
    }}
```

Features:

- Type-safe resolution
- Factory pattern support
- Lazy instantiation

9. Create a custom SwiftUI property wrapper for validation

Implementation:

```
@propertyWrapper
struct Validated {
    private var value: T
    private let validator: (T) -> Bool

    var wrappedValue: T {
        get { value }
        set { value = validator(newValue) ? newValue : value }
    }}
```

Advantages:

- Reusable validation logic
- Type-safe implementation
- SwiftUI compatible

10. Implement a custom Combine operator for debouncing with timeout

Solution:

```
extension Publishers {
    struct Debounce: Publisher {
        typealias Output = Upstream.Output
        typealias Failure = Upstream.Failure
        let upstream: Upstream
        let dueTime: DispatchQueue.SchedulerTimeType.Stride
    }}
```

Features:

- Custom operator implementation
- Configurable timing
- Type-safe publisher chain

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a thread-safe LRU Cache in Swift using generics

Solution:

Here's an efficient thread-safe LRU Cache implementation:

```
class LRUCache {
    private let capacity: Int
    private var cache = [K: V]()
    private var order = [K]()
    private let lock = NSLock()

    init(capacity: Int) { self.capacity = capacity }

    func get(_ key: K) -> V? {
        lock.lock(); defer { lock.unlock() }
        guard let value = cache[key] else { return nil }
        order.removeAll { $0 == key }
        order.append(key)
        return value
    }}
```

Key points:

- Uses NSLock for thread safety
- O(1) average time complexity for get/set operations
- Generic implementation supports any hashable key type

2. Implement a solution to find all pairs in an array that sum to a target value with O(n) time complexity

Solution:

```
func findPairs(_ array: [Int], target: Int) -> [(Int, Int)] {
    var seen = Set()
    var pairs = [(Int, Int)]()

    for num in array {
        let complement = target - num
        if seen.contains(complement) {
            pairs.append((complement, num))
        }
        seen.insert(num)
    }
    return pairs
}
```

Time Complexity: O(n) using a HashSet for constant-time lookups

Space Complexity: O(n) to store the seen numbers

3. Implement a thread-safe Singleton pattern in Swift

Solution:

```
class Singleton {
```

```
    static let shared = Singleton()
    private let queue = DispatchQueue(label: "com.singleton.queue")
    private var data: [String: Any] = [:]

    private init() {}

    func setValue(_ value: Any, forKey key: String) {
        queue.async { self.data[key] = value }
    }}
```

Key features:

- Thread-safe implementation using DispatchQueue
- Private initializer prevents external instantiation
- Static constant ensures single instance

4. Implement a sliding window algorithm to find the maximum sum subarray of size k

Solution:

```
func maxSumSubarray(_ array: [Int], _ k: Int) -> Int {
    guard array.count >= k else { return 0 }
    var maxSum = array[..
```

Time Complexity: O(n)

Space Complexity: O(1)

5. Implement a concurrent queue data structure in Swift

Solution:

```
class ConcurrentQueue {
    private var array = [T]()
    private let lock = NSLock()

    func enqueue(_ element: T) {
        lock.lock(); defer { lock.unlock() }
        array.append(element)
    }

    func dequeue() -> T? {
        lock.lock(); defer { lock.unlock() }
        return array.isEmpty ? nil : array.removeFirst()
    }}
```

Features:

- Thread-safe operations
- Generic implementation
- FIFO ordering

6. Implement a binary search tree with insertion and search operations in Swift

Solution:

```
class BST {
    class Node {
        var value: T
        var left, right: Node?
        init(_ value: T) { self.value = value }
    }
    private var root: Node?

    func insert(_ value: T) {
        root = insert(root, value)
    }

    private func insert(_ node: Node?, _ value: T) -> Node {
```

```
        guard let node = node else { return Node(value) }
        if value < node.value { node.left = insert(node.left, value) }
        else { node.right = insert(node.right, value) }
        return node
    }}
```

7. Implement a custom Stack data structure with O(1) minimum element retrieval

Solution:

```
struct MinStack {
    private var stack: [(element: T, min: T)] = []

    mutating func push(_ element: T) {
        let min = stack.isEmpty ? element : Swift.min(element, stack.last!.min)
        stack.append((element, min))
    }

    mutating func pop() -> T? {
        stack.popLast()?.element
    }
```

Features:

- O(1) push/pop operations
- O(1) minimum element access
- Generic implementation

8. Implement a solution for finding the longest substring without repeating characters

Solution:

```
func lengthOfLongestSubstring(_ s: String) -> Int {
    var chars = [Character: Int]()
    var maxLength = 0, start = 0

    for (end, char) in s.enumerated() {
        if let prevIndex = chars[char] { start = max(start, prevIndex + 1) }
        maxLength = max(maxLength, end - start + 1)
        chars[char] = end
    }
    return maxLength
}
```

Time Complexity: O(n)

Space Complexity: O(min(m,n)) where m is charset size

9. Implement a thread-safe circular buffer in Swift

Solution:

```
class CircularBuffer {
    private var buffer: [T?]
    private var head = 0, tail = 0
    private let lock = NSLock()

    init(capacity: Int) {
        buffer = Array(repeating: nil, count: capacity)
    }

    func write(_ element: T) -> Bool {
        lock.lock(); defer { lock.unlock() }
        if isFull { return false }
        buffer[tail] = element
        tail = (tail + 1) % buffer.count
        return true
    }}
```

Features:

- Fixed-size circular implementation
- Thread-safe operations
- O(1) read/write operations

10. Implement a trie (prefix tree) data structure for efficient string operations

Solution:

```
class Trie {
    class TrieNode {
        var children = [Character: TrieNode]()
        var isEnd = false
    }
    private let root = TrieNode()

    func insert(_ word: String) {
        var node = root
        for char in word {
            if node.children[char] == nil {
                node.children[char] = TrieNode()
            }
            node = node.children[char]!
        }
        node.isEnd = true
    }
```

Time Complexity:

- Insert: O(m) where m is word length
- Search: O(m) where m is word length
- Space: O(ALPHABET_SIZE * m * n) for n words

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly

Key Components & Considerations:

- **API Gateway**: Handle incoming requests for URL shortening and redirection
- **Hash Generation**: Create unique short codes using MD5/SHA256 + Base62 encoding
- **Data Storage**: NoSQL (like DynamoDB) for URL mappings
- **Cache Layer**: Redis/Memcached for frequently accessed URLs

Example Hash Generation in Swift:

```
func generateShortURL(_ longURL: String) -> String {
    let hash = longURL.data(using: .utf8)?.md5().base62
    return String(hash.prefix(7))
}
```

Scale Considerations:

- Load balancers for horizontal scaling
- CDN for global access
- Rate limiting to prevent abuse
- Analytics service for tracking clicks

2. Design a real-time chat system supporting millions of concurrent users

Architecture Components:

- **WebSocket Server**: For real-time bidirectional communication
- **Message Queue**: RabbitMQ/Kafka for message handling
- **Presence Service**: Track online/offline status
- **Storage**: MongoDB for messages, Redis for session management

Swift WebSocket Handler Example:

```
class ChatHandler: WebSocketDelegate {
    func didReceive(message: String) {
        broadcast(message)
        store(message)
        updatePresence()
    }
}
```

Scaling Strategy:

- Microservices architecture
- Sharding by chat room/user
- Message persistence with TTL

3. Design an Instagram-like social media feed system

Core Components:

- **Content Delivery**: CDN for media storage
- **Feed Service**: Aggregates posts for users
- **Graph Service**: Manages follower relationships

- **Cache Layer**: Redis for feed caching

Feed Generation Example:

```
struct FeedService {
    func generateFeed(for userID: String) async throws -> [Post] {
        let following = await graphService.getFollowing(userID)
        return await postService.getFeed(following, limit: 50)
    }
}
```

Optimization Techniques:

- Fan-out on write vs read
- Pagination with cursor-based approach
- Content preloading

4. Design a distributed task scheduler system

System Components:

- **Task Queue**: Redis/RabbitMQ for job storage
- **Worker Nodes**: Process tasks independently
- **Coordinator**: Manages task distribution
- **Monitor**: Tracks task status and retries

Task Processing Example:

```
class TaskWorker {
    func processTask(_ task: Task) async throws {
        guard let result = await executeTask(task)
        else { throw RetryError() }
        await updateTaskStatus(task.id, result)
    }
}
```

Reliability Features:

- At-least-once delivery
- Dead letter queues
- Task prioritization

5. Design a distributed caching system like Redis

Key Components:

- **Cache Nodes**: Store key-value pairs
- **Consistent Hashing**: Data distribution
- **Replication**: Master-slave setup
- **Eviction Policies**: LRU/LFU implementation

Cache Implementation Example:

```
class DistributedCache {
    func set(_ key: String, _ value: Any, ttl: TimeInterval) {
        let node = consistentHash.getNode(for: key)
        node.store(key, value, expiry: Date() + ttl)
    }
}
```

Scaling Features:

- Partition tolerance
- Hot key detection
- Cache warming strategies

6. Design a real-time analytics processing pipeline

Pipeline Components:

- **Event Collector**: Kafka/Kinesis
- **Stream Processor**: Spark/Flink
- **Storage Layer**: ClickHouse/Druid
- **Query Engine**: For real-time analytics

Event Processing Example:

```
struct AnalyticsProcessor {
    func processEvent(_ event: Event) async {
        let enrichedData = await enrich(event)
        await streamProcessor.process(enrichedData)
        await metricsAggregator.update(enrichedData)
    }
}
```

Scalability Features:

- Windowed aggregations
- Exactly-once processing
- Back-pressure handling

7. Design a distributed rate limiter

Architecture Components:

- **Token Bucket**: Rate limiting algorithm
- **Redis**: Distributed counter storage
- **Configuration Service**: Limit management
- **Monitor**: Usage tracking

Rate Limiter Implementation:

```
class RateLimiter {
    func allowRequest(userID: String) async -> Bool {
        let count = await redis.incr(userID)
        return count <= rateLimit
    }
}
```

Key Features:

- Sliding window algorithm
- Multi-tenant support
- Dynamic rate adjustment

8. Design a notification delivery system

System Components:

- **Push Gateway**: APNS/FCM integration
- **Template Engine**: Message customization
- **Delivery Service**: Handles retries
- **Preference Center**: User settings

Notification Handler Example:

```
struct NotificationService {
    func send(_ notification: Notification) async throws {
        let template = await templateEngine.render(notification)
        let devices = await getUserDevices(notification.userID)
        try await pushGateway.send(template, to: devices)
    }
}
```

Reliability Features:

- Delivery guarantees
- Rate limiting per user
- Batching capabilities

9. Design a distributed configuration management system

Core Components:

- **Config Store**: ZooKeeper/etcd
- **Change Propagation**: Pub/sub system
- **Version Control**: Config history
- **Access Control**: RBAC system

Config Client Example:

```
class ConfigClient {
    func watchConfig(_ key: String) async throws {
        for try await change in configStore.watch(key) {
            await updateLocalConfig(change)
        }
    }
}
```

Key Features:

- Dynamic updates
- Configuration validation
- Rollback support

10. Design a distributed logging system

System Components:

- **Log Collector**: Fluentd/Logstash
- **Message Queue**: Kafka for buffering
- **Storage**: Elasticsearch
- **Search API**: Query interface

Log Processing Example:

```
struct LogProcessor {
    func processLog(_ log: Log) async throws {
        let enriched = await addMetadata(log)
        try await store.index(enriched)
        await metrics.update(enriched)
    }
}
```

Scale Features:

- Log rotation/retention
- Index management
- Search optimization

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Swift function to check if a string is a palindrome, handling special characters and case sensitivity

Solution:

```
func isPalindrome(_ str: String) -> Bool {
    let cleaned = str.lowercased().filter { $0.isLetter }
    return cleaned == String(cleaned.reversed())
}
```

Key points:

- Removes special characters using filter
- Handles case sensitivity with lowercased()
- Uses reversed() for efficient comparison

2. Implement a function to flatten a nested array in Swift without using built-in flatten()

Solution:

```
func flatten(_ array: [Any]) -> [T] {
    var result = [T]()
    for element in array {
        if let arr = element as? [Any] { result += flatten(arr) }
        else if let value = element as? T { result.append(value) }
    }
    return result
}
```

Key aspects:

- Uses generics for type safety
- Handles recursive nesting
- Maintains type checking

3. Create a thread-safe singleton pattern in Swift

Solution:

```
class Singleton {
    static let shared = Singleton()
    private init() {}

    let queue = DispatchQueue(label: "singleton.queue")
    private var data: [String: Any] = [:]

    func setValue(_ value: Any, forKey key: String) {
        queue.async { self.data[key] = value }
    }
}
```

Important features:

- Uses static let for thread-safe initialization
- Implements DispatchQueue for thread-safe operations
- Private initializer prevents external instantiation

4. Write a function to detect and break retain cycles in closures

Example Implementation:

```
class DataManager {
    var completionHandler: (() -> Void)?

    func processData() {
        completionHandler = { [weak self] in
            guard let self = self else { return }
            self.handleCompletion()
        }
    }
}
```

Key concepts:

- Uses [weak self] to prevent retain cycles
- Implements guard let for safe unwrapping
- Demonstrates proper closure memory management

5. Implement a custom memory leak detector using Swift

Implementation:

```
class LeakDetector {
    static func track(_ object: AnyObject) {
        weak var weakRef = object
        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            assert(weakRef == nil, "Potential memory leak detected")
        }
    }
}
```

Features:

- Uses weak references for tracking
- Implements delayed checking
- Provides debugging assertions

6. Create a concurrent map operation using DispatchGroup

Solution:

```
extension Array {
    func concurrentMap(_ transform: @escaping (Element) -> T) -> [T] {
        let group = DispatchGroup()
        var result = [T?](repeating: nil, count: count)
        DispatchQueue.concurrentPerform(iterations: count) { i in
            group.enter()
            result[i] = transform(self[i])
            group.leave()
        }
        group.wait()
        return result.compactMap { $0 }
    }
```

7. Implement a custom property wrapper for UserDefaults storage

Implementation:

```
@propertyWrapper
struct UserDefault {
    let key: String
    let defaultValue: T

    var wrappedValue: T {
        get { UserDefaults.standard.object(forKey: key) as? T ?? defaultValue }
        set { UserDefaults.standard.set(newValue, forKey: key) }
    }
```

```
}
```

Usage:

- Simplifies UserDefaults access
- Provides type safety
- Supports default values

8. Write a function to safely unwrap multiple optionals

Solution:

```
func unwrap(_ opt1: T1?, _ opt2: T2?,
    _ transform: (T1, T2) -> Result) -> Result? {
    guard let value1 = opt1, let value2 = opt2 else { return nil }
    return transform(value1, value2)
}
```

Benefits:

- Type-safe unwrapping
- Reduces pyramid of doom
- Supports transformation

9. Implement a custom debug description protocol

Implementation:

```
extension CustomDebugStringConvertible {
    var debugDescription: String {
        let mirror = Mirror(reflecting: self)
        return mirror.children.map { "\($0.label ?? ""):\($0.value)" }
            .joined(separator: ", ")
    }
}
```

Features:

- Uses Mirror for reflection
- Provides formatted output
- Handles optional labels

10. Create a result builder for custom DSL implementation

Solution:

```
@resultBuilder
struct ArrayBuilder {
    static func buildBlock(_ components: T...) -> [T] {
        components
    }
    static func buildOptional(_ component: [T]?) -> [T] {
        component ?? []
    }
}
```

Applications:

- Custom DSL creation
- Declarative syntax support
- Type-safe builders