Android

Interview Questions and Answers

Core Concepts

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

1. Explain the Android Activity lifecycle and how it differs from Fragment lifecycle. What are the implications for state management?

Activity vs Fragment Lifecycle

Activity Lifecycle consists of onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy(). Activities are managed by the system and can be destroyed to reclaim memory.

Fragment Lifecycle includes additional callbacks: onAttach(), onCreateView(), onViewCreated(), onDestroyView(), onDetach(). Fragments are hosted within Activities and their lifecycle is tied to the host.

Key Differences

- **View lifecycle separation:** Fragments have onCreateView/onDestroyView allowing view recreation without Fragment recreation
- **Multiple instances:** Multiple Fragments can exist in one Activity simultaneously
- **Back stack behavior:** Fragment transactions can be added to back stack independently
- **Retained instances:** Fragments can be retained across configuration changes using setRetainInstance()

State Management Implications

Use **ViewModel** to survive configuration changes for both. For Fragments, be aware that **onDestroyView** is called during configuration changes while the Fragment instance persists, so clean up view references to avoid memory leaks. Always use **viewLifecycleOwner** for observers in Fragments instead of the Fragment's lifecycle.

2. What is the difference between LaunchMode attributes (standard, singleTop, singleTask, singleInstance)? Provide real-world use cases.

Launch Modes Overview

standard: Default mode. Creates a new instance every time the Activity is launched, even if one exists.

```
<activity android:launchMode="standard" />
```

singleTop: Reuses existing instance if it's at the top of the stack. Calls onNewIntent() instead of onCreate().

singleTask: Only one instance exists in the system. If instance exists, clears all Activities above it and calls onNewIntent().

singleInstance: Like singleTask, but the Activity runs in its own task with no other Activities.

Real-World Use Cases

- **standard:** Regular screens like detail pages, forms
- **singleTop:** Search screens, notification handlers where you don't want duplicate instances when already viewing
- **singleTask:** Main/home screens, email inbox - central navigation points
- **singleInstance:** Launcher apps, lock screens, video call screens that need complete isolation

Consider using **taskAffinity** with singleTask to control which task the Activity belongs to.

3. Explain the difference between Serializable and Parcelable. Why is Parcelable preferred in Android?

Serializable vs Parcelable

Serializable is a Java standard interface using reflection to serialize objects. Simple to implement (just implement the marker interface) but slower.

Parcelable is Android-specific, requiring explicit serialization logic but offering significantly better performance.

Parcelable Implementation

```
@Parcelize
data class User(
  val id: Int,
  val name: String
) : Parcelable
```

Why Parcelable is Preferred

- **Performance:** 10x faster than Serializable - critical for Android's IPC and Bundle operations
- **No reflection:** Explicit serialization avoids reflection overhead
- **Memory efficient:** Creates fewer temporary objects during serialization
- **Android optimized:** Designed specifically for Android's Binder IPC mechanism

With Kotlin's **@Parcelize** annotation, implementation is as simple as Serializable while maintaining performance benefits. Use Serializable only for persistence to disk or network transmission where compatibility matters.

4. How does Android's memory management work? Explain the Low Memory Killer and how to handle onTrimMemory() callback.

Android Memory Management

Android uses a **managed memory model** with automatic garbage collection. Each app runs in its own process with a limited heap size. The system kills processes when memory is low based on priority.

Low Memory Killer (LMK)

The LMK daemon monitors memory pressure and kills processes in this order:

- **Empty processes:** No active components
- **Background processes:** Activities in onStop()
- **Service processes:** Running services
- **Visible processes:** Visible but not foreground
- **Foreground processes:** Currently interacting with user

Handling onTrimMemory()

```
override fun onTrimMemory(level: Int) {
  super.onTrimMemory(level)
  when(level) {
    TRIM_MEMORY_RUNNING_CRITICAL ->
      clearImageCache()
    TRIM_MEMORY_UI_HIDDEN ->
      releaseUIResources()
  }
}
```

Best Practices

- Release cached data when receiving TRIM_MEMORY_RUNNING_CRITICAL
- Clear UI resources on TRIM_MEMORY_UI_HIDDEN
- Use WeakReference for caches
- Implement proper bitmap management with BitmapFactory.Options
- Monitor memory with Android Profiler

5. What are the differences between Service, IntentService, JobScheduler, and WorkManager? When should each be used?

Background Work Components

Service: Runs on main thread, requires manual thread management. Use for foreground services (music playback, navigation).

```
class MusicService : Service() {
  override fun onStartCommand(
    intent: Intent?,
    flags: Int,
    startId: Int
  ): Int {
    startForeground(ID, notification)
    return START_STICKY
  }
}
```

IntentService: Deprecated. Handled work on background thread sequentially. Stopped automatically when work completed.

JobScheduler: System service for scheduling deferred work with constraints (network, charging). API 21+.

WorkManager: Jetpack library wrapping JobScheduler/AlarmManager. Recommended for all deferrable background work.

When to Use Each

- **Foreground Service:** User-visible work (music, fitness tracking, downloads)
- **JobScheduler:** Direct API access needed, no backward compatibility required
- **WorkManager:** Guaranteed execution, backward compatible, constraint-based work (sync, uploads, cleanup)
- **Kotlin Coroutines:** Short async operations tied to lifecycle

WorkManager is the **recommended solution** for most background work scenarios.

6. Explain Content Providers and their role in Android. How do you implement custom Content Providers with proper URI matching?

Content Providers Overview

Content Providers manage access to structured data sets, enabling secure data sharing between applications. They abstract data storage (database, files, network) behind a consistent URI-based interface.

Custom Content Provider Implementation

```
class UserProvider : ContentProvider() {
  companion object {
    const val AUTHORITY = "com.app.users"
    val USERS_URI = Uri.parse(
      "content://$AUTHORITY/users")
    const val USERS = 1
    const val USER_ID = 2
  }
}
```

URI Matcher Setup

```
private val uriMatcher = UriMatcher(
  UriMatcher.NO_MATCH).apply {
  addURI(AUTHORITY, "users", USERS)
  addURI(AUTHORITY, "users/#", USER_ID)
}
```

Key Components

- **onCreate():** Initialize data sources
- **query():** Return Cursor with requested data
- **insert():** Add new records, return URI
- **update()/delete():** Modify data, return affected rows
- **getType():** Return MIME type for URI

Use Cases

- Sharing data between apps (contacts, calendar)
- Exposing database to other components
- Implementing sync adapters
- File sharing with FileProvider

7. What is the Android Binder framework? Explain how IPC (Inter-Process Communication) works in Android.

Android Binder Framework

Binder is Android's custom IPC mechanism enabling communication between processes. It's more efficient than traditional Linux IPC methods and provides security through process isolation.

How Binder IPC Works

- **Binder Driver:** Kernel module managing IPC transactions
- **Service Manager:** System service acting as name registry for Binder services
- **Proxy/Stub Pattern:** Client uses proxy, service implements stub
- **Parcel:** Message container for data serialization

AIDL Example

```
// IUserService.aidl
interface IUserService {
  User getUser(int id);
  void updateUser(in User user);
}
```

Service Implementation

```
class UserService : Service() {
  private val binder = object :
    IUserService.Stub() {
    override fun getUser(id: Int) =
      repository.getUser(id)
  }
}
```

Key Concepts

- **Synchronous by default:** Blocks caller until complete
- **oneway keyword:** Makes calls asynchronous
- **Transaction limit:** 1MB buffer size for all active transactions
- **Security:** Process isolation with permission checking

Used extensively in Android system services (ActivityManager, PackageManager).

8. Explain the difference between cold flow and hot flow in Kotlin. How do StateFlow and SharedFlow differ from Flow?

Cold vs Hot Flows

Cold Flow: Starts emitting only when collected. Each collector gets its own independent flow execution. Regular Flow is cold.

```
val coldFlow = flow {
  emit(fetchData()) // Called per collector
}.flowOn(Dispatchers.IO)
```

Hot Flow: Emits values regardless of collectors. Multiple collectors share the same flow execution.

StateFlow and SharedFlow are hot.

StateFlow

Hot flow that holds a **single current value**. Always has a value and emits the current state immediately to new collectors.

```
val _uiState = MutableStateFlow(
  UiState.Loading
)
val uiState: StateFlow<UiState> =
  _uiState.asStateFlow()
```

SharedFlow

Hot flow that can emit values to multiple collectors without holding state. Configurable replay cache and buffer.

```
val events = MutableSharedFlow<Event>(
  replay = 0,
  extraBufferCapacity = 64
)
```

When to Use Each

- **Flow:** One-shot operations, transformations
- **StateFlow:** UI state, always need current value
- **SharedFlow:** Events, multiple subscribers, no initial value needed

9. What is Jetpack Compose's recomposition? How does the Snapshot system work and what are best practices to optimize recomposition?

Recomposition in Jetpack Compose

Recomposition is the process of re-executing composable functions when their input state changes. Compose uses a smart diffing system to only recompose affected parts of the UI tree.

Snapshot System

The **Snapshot system** tracks state changes and schedules recomposition. It uses a copy-on-write mechanism to maintain state consistency during composition.

- State reads are recorded during composition
- When state changes, only composables that read that state recompose
- Changes are batched and applied atomically

Optimization Best Practices

```
@Composable
fun UserList(users: List<User>) {
  LazyColumn {
    items(
      items = users,
      key = { it.id } // Stable keys
    ) { user ->
      UserItem(user)
    }
  }
}
```

Key Optimization Techniques

- **Stable types:** Use @Stable or @Immutable annotations for data classes
- **Remember expensive operations:** Use remember { } for computed values
- **Derivedstateof:** For computed state from other state
- **Keys in lists:** Provide stable keys for list items
- **Lambda stability:** Extract lambdas to avoid recreation
- **Avoid reading state unnecessarily:** Only read state where needed

10. Explain Android's navigation component architecture. How does it handle deep links, safe args, and back stack management?

Navigation Component Architecture

The **Navigation Component** is a Jetpack library providing a framework for in-app navigation with a single Activity architecture. It consists of:

- **Navigation Graph:** XML resource defining all destinations and actions
- **NavHost:** Container displaying destinations
- **NavController:** Manages navigation within NavHost

Navigation Graph Example

```
<navigation
  app:startDestination="@id/home">
  <fragment android:id="@+id/home"
    android:name="HomeFragment">
    <action android:id="@+id/to_detail"
      app:destination="@id/detail"/>
  </fragment>
</navigation>
```

Deep Links

Define deep links in navigation graph with automatic handling:

```
<deepLink app:uri="myapp://detail/{id}"/>
```

Safe Args

Type-safe argument passing using generated classes:

```
val action = HomeFragmentDirections
  .toDetail(userId = 123)
findNavController().navigate(action)
```

Back Stack Management

- **popUpTo:** Pop destinations up to specified destination
- **popUpToInclusive:** Include the specified destination in pop
- **singleTop:** Avoid duplicate destinations on stack

Data Structures and Algorithms

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

1. Implement an LRU Cache in Kotlin with O(1) time complexity for get and put operations.

LRU Cache requires a combination of **HashMap** for O(1) lookup and a **Doubly Linked List** for O(1) insertion/deletion.

Implementation:

```
class LRUCache(private val capacity: Int) {
    private val map = LinkedHashMap(capacity, 0.75f, true)
    fun get(key: Int): Int = map.getOrDefault(key, -1)
    fun put(key: Int, value: Int) {
        map[key] = value
        if (map.size > capacity) map.remove(map.keys.first())
    }
}
```

Time Complexity: O(1) for both operations **Space Complexity:** O(capacity)

2. How would you find all pairs in an array that sum to a target value? What's the optimal approach?

Pair Sum Problem can be solved optimally using a **HashSet** in a single pass.

Optimal Solution:

```
fun findPairs(arr: IntArray, target: Int): List> {
    val seen = mutableSetOf()
    val pairs = mutableListOf>()
    arr.forEach { num ->
        if (seen.contains(target - num)) pairs.add(Pair(target - num, num))
        seen.add(num)
    }
    return pairs
}
```

Time Complexity: O(n) **Space Complexity:** O(n)

3. Implement a Stack that supports getMin() in O(1) time complexity.

Min Stack requires tracking the minimum value at each state using an auxiliary stack or storing pairs.

Implementation:

```
class MinStack {
    private val stack = Stack>()
    fun push(x: Int) {
        val min = if (stack.isEmpty()) x else minOf(x, stack.peek().second)
        stack.push(Pair(x, min))
    }
    fun pop() = stack.pop().first
    fun top() = stack.peek().first
    fun getMin() = stack.peek().second
}
```

Time Complexity: O(1) for all operations **Space Complexity:** O(n)

4. Explain the sliding window technique and implement a solution to find the maximum sum of k consecutive elements.

Sliding Window is an optimization technique that reduces nested loops to a single pass by maintaining a window of elements.

Maximum Sum of K Elements:

```
fun maxSumK(arr: IntArray, k: Int): Int {
    var maxSum = arr.take(k).sum()
    var windowSum = maxSum
    for (i in k until arr.size) {
        windowSum += arr[i] - arr[i - k]
        maxSum = maxOf(maxSum, windowSum)
    }
    return maxSum
}
```

Time Complexity: O(n) **Space Complexity:** O(1)

5. How do you detect a cycle in a linked list? Provide the Floyd's Cycle Detection algorithm implementation.

Floyd's Cycle Detection (Tortoise and Hare) uses two pointers moving at different speeds to detect cycles.

Implementation:

```
fun hasCycle(head: ListNode?): Boolean {
    var slow = head
    var fast = head
    while (fast?.next != null) {
        slow = slow?.next
        fast = fast.next?.next
        if (slow == fast) return true
    }
    return false
}
```

Time Complexity: O(n) **Space Complexity:** O(1)

6. Implement a Trie (Prefix Tree) for efficient word search operations in Android autocomplete scenarios.

Trie is ideal for prefix-based searches with O(m) complexity where m is the word length.

Basic Trie Implementation:

```
class TrieNode {
    val children = mutableMapOf()
    var isEndOfWord = false
}
class Trie {
    private val root = TrieNode()
    fun insert(word: String) {
        var node = root
        word.forEach { node = node.children.getOrPut(it) { TrieNode() } }
        node.isEndOfWord = true
    }
}
```

Time Complexity: O(m) for insert/search **Space Complexity:** O(n*m)

7. What's the difference between ArrayList and LinkedList in terms of performance? When would you use each in Android?

Performance Comparison:

- **ArrayList:** O(1) random access, O(n) insertion/deletion at arbitrary positions, backed by dynamic array
- **LinkedList:** O(n) random access, O(1) insertion/deletion at ends, O(n) at arbitrary positions

Android Use Cases:

- **ArrayList:** RecyclerView data sources, frequent index-based access, iteration-heavy operations
- **LinkedList:** Queue implementations, frequent additions/removals at ends, undo/redo mechanisms

General Rule: Prefer ArrayList in Android due to better cache locality and lower memory overhead unless you specifically need frequent insertions/deletions at both ends.

8. Implement a function to find the kth largest element in an unsorted array. What's the most efficient approach?

Quickselect Algorithm provides O(n) average time complexity, better than sorting O(n log n).

Using Min Heap (Practical Approach):

```
fun findKthLargest(nums: IntArray, k: Int): Int {
    val minHeap = PriorityQueue()
    nums.forEach { num ->
        minHeap.offer(num)
        if (minHeap.size > k) minHeap.poll()
    }
    return minHeap.peek()
}
```

Time Complexity: O(n log k) **Space Complexity:** O(k) **Alternative:** Quickselect offers O(n) average, O(n^2) worst case

9. How would you implement a thread-safe Singleton pattern using Kotlin for an Android database manager?

Thread-Safe Singleton in Kotlin can be elegantly implemented using the **object** keyword or **lazy delegation with synchronized**.

Best Practice Implementation:

```
class DatabaseManager private constructor(context: Context) {
    companion object {
        @Volatile private var instance: DatabaseManager? = null
        fun getInstance(context: Context) = instance ?: synchronized(this) {
            instance ?: DatabaseManager(context).also { instance = it }
        }
    }
}
```

Key Points: @Volatile ensures visibility, synchronized prevents race conditions, double-checked locking optimizes performance

10. Explain Binary Search Tree operations and implement a function to validate if a binary tree is a valid BST.

BST Validation requires checking that all left descendants < node < all right descendants, not just immediate children.

Correct Implementation:

```
fun isValidBST(root: TreeNode?, min: Long = Long.MIN_VALUE,
               max: Long = Long.MAX_VALUE): Boolean {
    if (root == null) return true
    if (root.`val` <= min || root.`val` >= max) return false
    return isValidBST(root.left, min, root.`val`.toLong()) &&
           isValidBST(root.right, root.`val`.toLong(), max)
}
```

Time Complexity: O(n) **Space Complexity:** O(h) where h is tree height

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 for Android. What are the key components and how would you handle billions of URLs?

Architecture Overview

A URL shortener requires careful consideration of **scalability**, **high availability**, and **low latency**.

Key Components

- **API Gateway:** Handle incoming requests with rate limiting
- **Application Servers:** Stateless servers for horizontal scaling
- **Database:** NoSQL (Cassandra/DynamoDB) for write-heavy workload
- **Cache Layer:** Redis for hot URLs (80-20 rule)
- **CDN:** Serve static content and reduce latency

URL Generation Strategy

Use **Base62 encoding** (a-z, A-Z, 0-9) to generate 7-character short URLs, providing 62^7 = 3.5 trillion combinations.

```
fun generateShortUrl(id: Long): String {
    val base62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    var num = id
    val sb = StringBuilder()
    while (num > 0) {
        sb.append(base62[(num % 62).toInt()])
        num /= 62
    }
    return sb.reverse().toString().padStart(7, '0')
}
```

Database Schema

- **URLs Table:** id (primary key), original_url, short_url, created_at, expires_at
- **Analytics Table:** short_url, click_count, last_accessed, user_agent

Scalability Considerations

- **Sharding:** Partition by hash of short_url for distributed storage
- **Read Replicas:** Multiple read replicas for high read throughput
- **Caching:** Cache popular URLs with TTL-based invalidation
- **Async Processing:** Use message queues for analytics updates

CAP Theorem Trade-offs

Favor **Availability and Partition Tolerance (AP)** over strong consistency. Eventual consistency is acceptable for analytics, but URL mappings should be consistent.

2. Design a real-time chat application for Android. How would you handle message delivery, offline support, and synchronization across devices?

Architecture Components

- **WebSocket Server:** Persistent bidirectional connections for real-time messaging
- **Message Queue:** RabbitMQ or Kafka for reliable message delivery
- **Presence Service:** Track online/offline status using Redis

- **Push Notification Service:** FCM for offline message delivery
- **Storage:** Cassandra for message history, Redis for recent messages

Android Client Architecture

```
class ChatRepository(private val api: ChatApi, private val db: ChatDatabase) {
    suspend fun sendMessage(msg: Message) {
        db.insertMessage(msg.copy(status = PENDING))
        try {
            val sent = api.sendMessage(msg)
            db.updateMessage(sent.copy(status = SENT))
        } catch (e: Exception) {
            db.updateMessage(msg.copy(status = FAILED))
        }
    }
}
```

Message Delivery Guarantees

- **At-least-once delivery:** Use acknowledgment system with retry logic
- **Message IDs:** UUID for deduplication on client side
- **Sequence Numbers:** Maintain order per conversation

Offline Support Strategy

- **Local Database:** Room database for message persistence
- **Sync Queue:** WorkManager for background synchronization
- **Conflict Resolution:** Last-write-wins with server timestamp

Multi-Device Synchronization

- **Message Cursor:** Each device maintains last_sync_timestamp
- **Delta Sync:** Fetch only messages after cursor
- **Device Registration:** Register all devices with user account for push notifications

Scalability Patterns

- **Connection Pooling:** WebSocket servers behind load balancer with sticky sessions
- **Horizontal Scaling:** Stateless message handlers with shared state in Redis
- **Message Fanout:** Pub/Sub pattern for group chats

3. Design a social media feed system for Android (like Instagram or Twitter). How would you handle feed generation, ranking, and real-time updates?

Feed Architecture Types

Two main approaches: **Pull Model (Fan-out on read)** vs **Push Model (Fan-out on write)**

Hybrid Approach (Recommended)

- **Push for active users:** Pre-compute feeds for users who log in frequently
- **Pull for inactive users:** Generate feed on-demand for occasional users
- **Celebrity handling:** Use pull model for users with millions of followers

System Components

- **Post Service:** Handle post creation and storage
- **Feed Generation Service:** Background workers to build feeds
- **Feed Storage:** Redis sorted sets for timeline (user_id -> list of post_ids)
- **Ranking Service:** ML-based ranking for personalized feeds
- **CDN:** Serve media content (images/videos)

Android Client Implementation

```
class FeedViewModel : ViewModel() {
    private val _feed = MutableStateFlow>(PagingData.empty())
    val feed = _feed.asStateFlow()
    init {
        viewModelScope.launch {
            repository.getFeed().cachedIn(viewModelScope).collect {
                _feed.value = it
            }
        }
    }
}
```

Feed Ranking Algorithm

- **Engagement Score:** likes + comments * 2 + shares * 3
- **Recency:** Exponential decay based on post age
- **User Affinity:** Interaction history with post author
- **Content Type:** User preference for video vs images

Real-Time Updates

- **WebSocket Connection:** Push new posts to active users
- **Polling Fallback:** Periodic API calls for connection failures
- **Optimistic Updates:** Show user's own posts immediately

Caching Strategy

- **L1 Cache:** In-memory cache for current session
- **L2 Cache:** Room database for offline access
- **L3 Cache:** Redis on server for hot content

Scalability Considerations

- **Pagination:** Cursor-based pagination for infinite scroll
- **Sharding:** Partition feeds by user_id hash
- **Read Replicas:** Distribute read load across multiple databases

4. Design a distributed caching system for an Android app that works across multiple servers. How would you handle cache invalidation and consistency?

Cache Architecture Layers

- **Client Cache:** In-memory LRU cache on Android device
- **Edge Cache:** CDN for static assets and API responses
- **Distributed Cache:** Redis cluster for shared state
- **Database:** Source of truth for persistent data

Android Client Cache Implementation

```
class LruCache(maxSize: Int) : LinkedHashMap(16, 0.75f, true) {
    private val maxSize = maxSize

    override fun removeEldestEntry(eldest: Map.Entry): Boolean {
        return size > maxSize
    }

    fun getOrPut(key: K, loader: () -> V): V {
        return get(key) ?: loader().also { put(key, it) }
    }
}
```

Cache Invalidation Strategies

- **TTL-based:** Set expiration time for cache entries
- **Event-driven:** Invalidate on data mutations using pub/sub
- **Version-based:** Include version number in cache key
- **Write-through:** Update cache synchronously with database writes

Consistency Models

- **Strong Consistency:** Always read from source, use cache only for reads after writes
- **Eventual Consistency:** Accept stale reads, propagate updates asynchronously
- **Read-Your-Writes:** User sees their own updates immediately

Cache Coherence Protocol

- **Invalidate Pattern:** Remove cache entry on update, lazy reload on next read
- **Update Pattern:** Push updated value to all cache nodes
- **Refresh-Ahead:** Proactively refresh hot keys before expiration

Distributed Cache Challenges

- **Cache Stampede:** Use locking or probabilistic early expiration
- **Thundering Herd:** Implement request coalescing and jittered backoff
- **Hot Keys:** Replicate popular keys across multiple nodes

Redis Cluster Configuration

- **Sharding:** Consistent hashing for key distribution
- **Replication:** Master-slave for high availability
- **Persistence:** RDB snapshots + AOF for durability

5. Design a ride-sharing application like Uber for Android. How would you handle real-time location tracking, driver matching, and trip management?

System Architecture

- **Location Service:** Track and update driver/rider locations in real-time
- **Matching Service:** Find optimal driver-rider pairs
- **Trip Service:** Manage trip lifecycle (request, accept, start, complete)
- **Pricing Service:** Calculate dynamic pricing based on demand
- **Notification Service:** Push updates to drivers and riders

Location Tracking on Android

```
class LocationTracker(private val fusedClient: FusedLocationProviderClient) {
    fun startTracking() = callbackFlow {
        val request = LocationRequest.create().apply {
            interval = 5000
            priority = LocationRequest.PRIORITY_HIGH_ACCURACY
        }
        val callback = object : LocationCallback() {
            override fun onLocationResult(result: LocationResult) {
                trySend(result.lastLocation)
            }
        }
    }
}
```

Geospatial Indexing

- **Geohashing:** Divide map into grid cells for efficient proximity search
- **QuadTree:** Hierarchical spatial data structure for dynamic density
- **Google S2:** Spherical geometry for accurate distance calculations

Driver Matching Algorithm

- **Proximity Search:** Find drivers within radius using geohash prefix matching
- **Scoring Function:** distance * 0.7 + rating * 0.2 + acceptance_rate * 0.1
- **Timeout Mechanism:** Expand search radius if no match in 30 seconds
- **Concurrent Matching:** Prevent double-booking with distributed locks

Real-Time Communication

- **WebSocket:** Bidirectional communication for live updates
- **MQTT:** Lightweight protocol for IoT-like location updates
- **Server-Sent Events:** One-way streaming for trip status updates

Trip State Machine

- **States:** REQUESTED → ACCEPTED → ARRIVED → IN_PROGRESS → COMPLETED
- **Transitions:** Validated on server to prevent invalid state changes
- **Idempotency:** Use unique trip_id to handle duplicate requests

Scalability Patterns

- **Geo-Sharding:** Partition by geographic regions
- **Event Sourcing:** Store all trip events for audit and replay
- **CQRS:** Separate read/write models for different access patterns

6. Design a video streaming service for Android like YouTube. How would you handle video upload, transcoding, adaptive bitrate streaming, and CDN integration?

System Components

- **Upload Service:** Chunked upload with resumability
- **Transcoding Pipeline:** Convert videos to multiple formats and resolutions
- **Storage:** Object storage (S3) for video files
- **CDN:** CloudFront/Cloudflare for global content delivery
- **Metadata Service:** Store video info, thumbnails, subtitles

Video Upload on Android

```
class VideoUploader(private val api: UploadApi) {
    suspend fun upload(file: File, chunkSize: Int = 1024 * 1024) {
        val chunks = file.readBytes().toList().chunked(chunkSize)
        chunks.forEachIndexed { index, chunk ->
            api.uploadChunk(
                videoId = UUID.randomUUID().toString(),
                chunkIndex = index,
                data = chunk.toByteArray()
            )
        }
    }
}
```

Transcoding Pipeline

- **Input Queue:** SQS/Kafka for upload notifications
- **Worker Pool:** Auto-scaling EC2 instances running FFmpeg
- **Output Formats:** 360p, 480p, 720p, 1080p, 4K
- **Codecs:** H.264 for compatibility, H.265 for efficiency, VP9 for web

Adaptive Bitrate Streaming (ABR)

- **HLS Protocol:** Segment videos into .ts chunks with .m3u8 playlist
- **DASH Protocol:** ISO standard for adaptive streaming
- **Bitrate Ladder:** 500kbps, 1Mbps, 2.5Mbps, 5Mbps, 8Mbps

Android Player Implementation

```
class VideoPlayer(context: Context) {
    private val exoPlayer = ExoPlayer.Builder(context).build()

    fun play(url: String) {
        val mediaItem = MediaItem.fromUri(url)
        exoPlayer.setMediaItem(mediaItem)
        exoPlayer.prepare()
        exoPlayer.play()
    }
}
```

CDN Strategy

- **Origin Shield:** Additional cache layer to reduce origin load

- **Edge Locations:** Serve from nearest geographic location
- **Cache Control:** Long TTL for immutable video segments
- **Prefetching:** Predictively fetch next segments

Scalability Considerations

- **Thumbnail Generation:** Extract keyframes during transcoding
- **DRM Integration:** Widevine for content protection
- **Analytics:** Track watch time, buffering events, quality switches
- **Cost Optimization:** Use cheaper storage tiers for old content

7. Design a distributed task scheduler for Android background jobs. How would you ensure reliability, ordering, and handle failures?

Architecture Overview

- **Task Queue:** Persistent queue (Redis/RabbitMQ) for pending tasks
- **Scheduler Service:** Distribute tasks to workers
- **Worker Pool:** Execute tasks with retry logic
- **State Store:** Track task status and results
- **Dead Letter Queue:** Handle permanently failed tasks

Android WorkManager Integration

```
class SyncWorker(context: Context, params: WorkerParameters) : CoroutineWorker(context, params) {
    override suspend fun doWork(): Result {
        return try {
            repository.syncData()
            Result.success()
        } catch (e: Exception) {
            if (runAttemptCount < 3) Result.retry()
            else Result.failure()
        }
    }
}
```

Task Scheduling Strategies

- **Immediate:** Execute as soon as worker is available
- **Delayed:** Schedule for future execution with timestamp
- **Periodic:** Recurring tasks with fixed intervals
- **Conditional:** Execute only when constraints are met (network, battery)

Reliability Guarantees

- **At-least-once:** Retry failed tasks, handle idempotency on consumer side
- **At-most-once:** No retries, accept potential data loss
- **Exactly-once:** Use distributed transactions with 2PC or Saga pattern

Task Ordering

- **FIFO Queue:** Preserve insertion order for sequential processing
- **Priority Queue:** High-priority tasks execute first
- **Dependency Graph:** DAG for tasks with dependencies
- **Chaining:** WorkManager chains for ordered execution

Failure Handling

- **Exponential Backoff:** Increase delay between retries (1s, 2s, 4s, 8s)
- **Circuit Breaker:** Stop retrying if downstream service is down
- **Compensation:** Rollback actions for failed distributed transactions
- **Monitoring:** Alert on high failure rates or queue depth

Distributed Challenges

- **Split Brain:** Use consensus algorithms (Raft/Paxos) for leader election
- **Task Duplication:** Unique task IDs with deduplication window

- **Worker Crashes:** Heartbeat mechanism with task reassignment

8. Design a search and autocomplete system for an Android e-commerce app. How would you handle indexing, ranking, and real-time suggestions?

Search Architecture

- **Indexing Service:** Build inverted index from product catalog
- **Search Engine:** Elasticsearch or Solr for full-text search
- **Autocomplete Service:** Trie data structure for prefix matching
- **Ranking Service:** ML model for relevance scoring
- **Cache Layer:** Redis for popular queries

Android Autocomplete Implementation

```
class SearchViewModel : ViewModel() {
    val searchResults = searchQuery
        .debounce(300)
        .filter { it.length >= 2 }
        .distinctUntilChanged()
        .flatMapLatest { query ->
            repository.autocomplete(query)
        }
        .stateIn(viewModelScope, SharingStarted.Lazily, emptyList())
}
```

Indexing Strategy

- **Document Structure:** product_id, title, description, category, price, ratings
- **Analyzers:** Tokenization, stemming, stop word removal, synonyms
- **Sharding:** Partition index by category or hash for scalability
- **Replication:** Multiple replicas for high availability

Ranking Algorithm

- **TF-IDF:** Term frequency and inverse document frequency
- **BM25:** Probabilistic ranking function for relevance
- **Boosting:** Increase score for title matches, popular products
- **Personalization:** User history and preferences
- **Business Rules:** Promote sponsored products, in-stock items

Autocomplete Optimization

- **Prefix Trie:** Store popular queries in memory for O(k) lookup
- **Query Completion:** Suggest based on historical search data
- **Fuzzy Matching:** Handle typos with edit distance (Levenshtein)
- **Context-Aware:** Suggestions based on user location, time, season

Performance Optimization

- **Edge N-grams:** Index prefixes for faster autocomplete
- **Caching:** Cache top 10K queries covering 80% of traffic
- **Query Expansion:** Include synonyms and related terms
- **Result Pagination:** Return top 20 results, load more on scroll

Scalability Patterns

- **Read Replicas:** Distribute search load across multiple nodes
- **Geo-Distribution:** Deploy search clusters in multiple regions
- **Async Indexing:** Update index asynchronously via message queue

9. Design a payment processing system for an Android app. How would you ensure security, handle transactions, and maintain consistency?

System Architecture

- **Payment Gateway:** Integration with Stripe, PayPal, or custom processor

- **Transaction Service:** Manage payment lifecycle
- **Ledger Service:** Double-entry bookkeeping for financial records
- **Fraud Detection:** Real-time risk assessment
- **Notification Service:** Send payment confirmations

Android Payment Flow

```
class PaymentProcessor(private val stripe: Stripe) {
    suspend fun processPayment(amount: Int, token: String): Result {
        return try {
            val intent = stripe.createPaymentIntent(amount, "usd")
            val result = stripe.confirmPayment(intent.id, token)
            Result.success(result)
        } catch (e: StripeException) {
            Result.failure(e)
        }
    }
}
```

Security Measures

- **PCI DSS Compliance:** Never store raw card data on device or server
- **Tokenization:** Replace sensitive data with tokens
- **SSL/TLS:** Encrypt all data in transit
- **Certificate Pinning:** Prevent MITM attacks
- **3D Secure:** Two-factor authentication for card payments

Transaction States

- **PENDING:** Payment initiated, awaiting confirmation
- **AUTHORIZED:** Funds reserved, not yet captured
- **CAPTURED:** Funds transferred to merchant
- **FAILED:** Payment rejected by processor
- **REFUNDED:** Money returned to customer

Consistency Guarantees

- **Idempotency:** Use unique idempotency keys to prevent duplicate charges
- **Two-Phase Commit:** Coordinate across payment gateway and order service
- **Saga Pattern:** Compensating transactions for distributed rollback
- **Event Sourcing:** Store all payment events for audit trail

Fraud Detection

- **Velocity Checks:** Limit transaction frequency per user
- **Geolocation:** Flag mismatches between device and card location
- **Device Fingerprinting:** Track suspicious device patterns
- **ML Models:** Real-time scoring based on historical fraud patterns

Failure Handling

- **Retry Logic:** Exponential backoff for network failures
- **Circuit Breaker:** Fail fast if payment gateway is down
- **Reconciliation:** Daily batch job to match transactions with gateway
- **Manual Review:** Queue high-risk transactions for human verification

10. Design a notification system for an Android app that supports push notifications, in-app notifications, and email. How would you handle delivery, prioritization, and user preferences?

System Architecture

- **Notification Service:** Orchestrate notification delivery
- **Template Engine:** Generate notification content
- **Delivery Workers:** Send via FCM, email, SMS
- **Preference Service:** Store user notification settings
- **Analytics Service:** Track delivery and engagement metrics

Android FCM Integration

```
class NotificationService : FirebaseMessagingService() {
    override fun onMessageReceived(message: RemoteMessage) {
        val notification = NotificationCompat.Builder(this, CHANNEL_ID)
            .setContentTitle(message.notification?.title)
            .setContentText(message.notification?.body)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .build()
        notificationManager.notify(id, notification)
    }
}
```

Notification Types

- **Transactional:** Order confirmations, payment receipts (high priority)
- **Promotional:** Marketing campaigns, offers (low priority)
- **Social:** Likes, comments, follows (medium priority)
- **System:** App updates, security alerts (critical priority)

Delivery Strategy

- **Multi-Channel:** Send via push, email, SMS based on user preference
- **Fallback:** Try push first, fallback to email if device offline
- **Batching:** Aggregate similar notifications to reduce noise
- **Quiet Hours:** Respect user's do-not-disturb settings

Prioritization

- **Priority Queue:** Process high-priority notifications first
- **Rate Limiting:** Limit notifications per user per hour
- **Deduplication:** Prevent sending duplicate notifications
- **Expiration:** Discard stale notifications after TTL

User Preferences

- **Granular Controls:** Enable/disable by notification type
- **Channel Selection:** Choose push, email, SMS, or all
- **Frequency Capping:** Daily digest vs real-time
- **Opt-out:** Global unsubscribe option

Scalability Patterns

- **Message Queue:** Kafka for high-throughput notification processing
- **Worker Pool:** Horizontal scaling of delivery workers
- **Sharding:** Partition users across multiple queues
- **Caching:** Cache user preferences in Redis

Analytics & Monitoring

- **Delivery Metrics:** Sent, delivered, failed, bounced
- **Engagement:** Open rate, click-through rate, conversion
- **A/B Testing:** Test different notification content and timing

Coding and Debugging

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

1. Write a Kotlin function to flatten a nested list of integers.

Solution

Use recursion to traverse nested lists and collect all integers into a single flat list.

```
fun flattenList(list: List): List {
    val result = mutableListOf()
    for (item in list) {
        when (item) {
            is Int -> result.add(item)
            is List<*> -> result.addAll(flattenList(item as List))
        }
    }
    return result
}
```

Key Points:

- Uses type checking with **when** expression
- Recursively processes nested lists
- Returns a flattened List

2. How do you detect and fix memory leaks in Android applications?

Detection and Prevention

Detection Tools:

- **LeakCanary:** Automatically detects memory leaks in debug builds
- **Android Profiler:** Memory profiler in Android Studio to track allocations
- **Heap Dump Analysis:** Use Memory Profiler to capture and analyze heap dumps

Common Causes and Fixes:

- **Context leaks:** Use WeakReference or ApplicationContext instead of Activity context
- **Static references:** Avoid static references to Views or Activities
- **Anonymous inner classes:** Use static inner classes with WeakReference
- **Listeners not unregistered:** Always unregister listeners in onDestroy/onPause
- **Handler leaks:** Use static Handler with WeakReference to outer class

3. Write a function to check if a string is a palindrome, ignoring spaces and case.

Solution

Clean the string and compare it with its reverse.

```
fun isPalindrome(str: String): Boolean {
    val cleaned = str.replace("\\s".toRegex(), "")
                  .lowercase()
    return cleaned == cleaned.reversed()
}

// Alternative two-pointer approach
fun isPalindromeOptimized(str: String): Boolean {
    val cleaned = str.filter { it.isLetterOrDigit() }.lowercase()
    var left = 0
    var right = cleaned.length - 1
    while (left < right) {
        if (cleaned[left++] != cleaned[right--]) return false
    }
    return true
}
```

Time Complexity: O(n) for both approaches

4. What debugging tools does Android Studio provide and how do you use them effectively?

Essential Debugging Tools

1. Debugger:

- Breakpoints (line, conditional, exception)
- Step Over/Into/Out for code navigation
- Evaluate Expression to test code at runtime
- Watch variables to monitor value changes

2. Logcat:

- Filter by log level, tag, and package
- Use structured logging with Timber library
- Create custom filters for specific scenarios

3. Android Profiler:

- **CPU Profiler:** Trace method execution and identify bottlenecks
- **Memory Profiler:** Track allocations and detect leaks
- **Network Profiler:** Monitor network requests and responses
- **Energy Profiler:** Analyze battery consumption

4. Layout Inspector:

- View hierarchy in real-time
- Inspect view properties and dimensions
- Debug UI rendering issues

5. Implement a singleton pattern in Kotlin that is thread-safe and lazy-initialized.

Multiple Approaches

Approach 1: Object Declaration (Recommended)

```
object DatabaseManager {
    fun connect() { /* implementation */ }
}
```

Approach 2: Lazy with Synchronized

```
class DatabaseManager private constructor() {
    companion object {
        val instance: DatabaseManager by lazy(LazyThreadSafetyMode.SYNCHRONIZED) {
            DatabaseManager()
        }
    }
}
```

Approach 3: Double-Check Locking

```
class DatabaseManager private constructor() {
    companion object {
        @Volatile private var INSTANCE: DatabaseManager? = null
        fun getInstance() = INSTANCE ?: synchronized(this) {
            INSTANCE ?: DatabaseManager().also { INSTANCE = it }
        }
    }
}
```

6. How do you handle uncaught exceptions globally in an Android app?

Global Exception Handling

Implement a custom **UncaughtExceptionHandler** to catch and handle exceptions before the app crashes.

```
class GlobalExceptionHandler(private val context: Context) : Thread.UncaughtExceptionHandler {
    private val defaultHandler = Thread.getDefaultUncaughtExceptionHandler()

    override fun uncaughtException(thread: Thread, throwable: Throwable) {
        // Log to analytics
        FirebaseCrashlytics.getInstance().recordException(throwable)
        // Show error activity or restart app
        val intent = Intent(context, ErrorActivity::class.java)
        intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
        context.startActivity(intent)
        defaultHandler?.uncaughtException(thread, throwable)
    }
}
```

Registration in Application class:

```
class MyApp : Application() {
    override fun onCreate() {
        super.onCreate()
        Thread.setDefaultUncaughtExceptionHandler(GlobalExceptionHandler(this))
    }
}
```

7. What is StrictMode and how do you use it for debugging performance issues?

StrictMode Overview

StrictMode is a developer tool that detects accidental disk or network access on the main thread, and other policy violations.

Implementation:

```
if (BuildConfig.DEBUG) {
    StrictMode.setThreadPolicy(
        StrictMode.ThreadPolicy.Builder()
            .detectDiskReads()
            .detectDiskWrites()
            .detectNetwork()
            .detectCustomSlowCalls()
            .penaltyLog()
            .penaltyFlashScreen()
            .build()
    )
    StrictMode.setVmPolicy(
        StrictMode.VmPolicy.Builder()
            .detectLeakedSqlLiteObjects()
            .detectLeakedClosableObjects()
            .detectActivityLeaks()
            .penaltyLog()
            .build()
    )
}
```

Benefits:

- Identifies blocking I/O operations on main thread
- Detects memory leaks early
- Enforces best practices during development

8. Write a function to reverse words in a sentence while maintaining word order.

Solution

Split the sentence into words, reverse each word individually, then join them back.

```
fun reverseWords(sentence: String): String {
    return sentence.split(" ")
        .joinToString(" ") { it.reversed() }
}

// Example: "Hello World" -> "olleH dlroW"

// Alternative: Reverse entire sentence
fun reverseSentence(sentence: String): String {
    return sentence.split(" ").reversed().joinToString(" ")
}

// Example: "Hello World" -> "World Hello"
```

Time Complexity: O(n) where n is the length of the string

9. How do you profile and optimize app startup time in Android?

Startup Optimization Strategies

1. Measurement Tools:

- **adb command:** adb shell am start -W com.package.name/.MainActivity
- **Reportfully Drawn:** Call reportFullyDrawn() when UI is ready
- **Android Profiler:** Use CPU Profiler to trace startup methods
- **Macrobenchmark:** Automated startup performance testing

2. Optimization Techniques:

- **Lazy initialization:** Defer non-critical initialization
- **Background threads:** Move heavy operations off main thread
- **Content Provider:** Avoid initialization in ContentProvider.onCreate()
- **App Startup library:** Use Jetpack App Startup for organized initialization
- **Baseline Profiles:** Use baseline profiles for ART optimization
- **Reduce layout complexity:** Flatten view hierarchies

```
class MyApp : Application() {
    override fun onCreate() {
        super.onCreate()
        // Critical init only
        initCrashReporting()
        // Defer other init
        lifecycleScope.launch { initNonCritical() }
    }
}
```

10. Explain method count limitations and how to handle the 64K DEX limit.

DEX Limit Overview

A single DEX file can reference a maximum of **65,536 methods** (64K). When your app exceeds this, you need multidex.

Solution 1: Enable Multidex

```
// build.gradle
android {
    defaultConfig {
        multiDexEnabled true
    }
}

dependencies {
    implementation 'androidx.multidex:multidex:2.0.1'
}

// Application class
```

```
class MyApp : MultiDexApplication() {
    // or override attachBaseContext if extending Application
}
```

Solution 2: Reduce Method Count

- **ProGuard/R8:** Enable code shrinking to remove unused methods
- **Dependency review:** Remove unused libraries
- **Use lite versions:** e.g., play-services-maps instead of play-services
- **Avoid duplicate libraries:** Resolve version conflicts

Analysis: Use ./gradlew app:dependencies to analyze method count

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize an Android app's performance under tight deadlines.

Situation: Our e-commerce app was experiencing ANRs and slow load times during a major sale event, with user complaints increasing by 40%.

Task: I was tasked with identifying and resolving performance bottlenecks within 48 hours before the next promotional wave.

Action: I used Android Profiler to identify memory leaks in RecyclerView adapters, implemented image loading with Glide caching, moved database queries to background threads using Coroutines, and reduced overdraw by flattening view hierarchies.

Result: App launch time decreased by 35%, ANR rate dropped by 60%, and we successfully handled 3x traffic during the sale with positive user feedback.

2. Describe a situation where you had to resolve a critical production bug in an Android application.

Situation: A payment processing feature started crashing for 15% of users after a production release, causing revenue loss and negative reviews.

Task: I needed to quickly identify the root cause, implement a fix, and restore user confidence without rolling back the entire release.

Action: I analyzed Firebase Crashlytics logs and discovered a NullPointerException in the payment fragment due to improper lifecycle handling. I implemented a hotfix using proper null-safety checks and lifecycle-aware components, tested it thoroughly, and deployed via staged rollout.

Result: The crash rate dropped to 0.2% within 6 hours, we recovered 95% of affected users, and implemented additional automated testing to prevent similar issues.

3. Share an example of when you had to mentor junior developers on Android best practices.

Situation: Three junior developers joined our team and were struggling with Android architecture patterns, leading to code inconsistency and increased code review time.

Task: I was assigned to mentor them and establish coding standards that would improve code quality and team velocity.

Action: I created a comprehensive onboarding guide covering MVVM, Clean Architecture, and Jetpack components. I conducted weekly code review sessions, pair programming exercises, and created reusable code templates. I also established a knowledge-sharing Slack channel for quick questions.

Result: Within two months, code review iterations decreased by 50%, the juniors independently implemented three features using proper architecture, and they became active contributors to our internal Android library.

4. Tell me about a time when you had to make a difficult technical decision regarding app architecture.

Situation: Our legacy app built with MVP pattern was becoming unmaintainable with 200+ activities and tight coupling, making feature development slow and bug-prone.

Task: I needed to propose and lead an architectural migration strategy that wouldn't disrupt ongoing feature development or introduce regressions.

Action: I evaluated multiple approaches (complete rewrite vs. incremental migration), created a proof-of-concept using MVVM with Clean Architecture and Jetpack components, presented cost-benefit analysis to stakeholders, and designed a phased migration plan targeting high-churn modules first.

Result: We successfully migrated 60% of the app over 6 months, reduced bug count by 40%, improved test coverage from 30% to 75%, and decreased feature development time by 25%.

5. Describe a situation where you had to handle conflicting priorities from product and engineering teams.

Situation: Product wanted to ship a new social sharing feature within one week, but engineering identified critical technical debt in our networking layer that was causing intermittent failures.

Task: I needed to balance business needs with technical stability while maintaining team morale and code quality.

Action: I organized a stakeholder meeting presenting data on current failure rates and potential impact on user experience. I proposed a compromise: implement a simplified version of the social feature using existing stable components while allocating 40% of sprint capacity to networking refactoring. I created a detailed timeline showing how this approach would prevent future delays.

Result: Both initiatives were completed within two weeks, network reliability improved by 85%, the social feature launched successfully, and we established a new process for balancing feature work with technical improvements.

6. Tell me about a time when you improved the development workflow or CI/CD pipeline for Android.

Situation: Our Android build process took 45 minutes, blocking developers from getting quick feedback and slowing down the release cycle.

Task: I was tasked with optimizing the build system and CI/CD pipeline to improve developer productivity and enable faster releases.

Action: I implemented Gradle build caching, modularized the monolithic app into feature modules, configured incremental builds, set up parallel execution, and migrated CI to GitHub Actions with matrix builds. I also introduced snapshot testing and selective test execution based on changed modules.

Result: Build time reduced from 45 to 8 minutes, CI pipeline execution time decreased by 60%, developer satisfaction increased significantly, and we moved from bi-weekly to weekly releases with confidence.

7. Share an experience where you had to work with cross-functional teams to deliver a complex Android feature.

Situation: We needed to implement a real-time video consultation feature requiring coordination between Android, iOS, backend, design, and QA teams across three time zones.

Task: As the Android lead, I needed to ensure seamless integration, consistent user experience across platforms, and on-time delivery for a major product launch.

Action: I established weekly sync meetings, created a shared technical specification document, set up a unified API contract with backend team, coordinated with iOS for feature parity, worked closely with designers on Android-specific UI adaptations, and implemented WebRTC with proper error handling and network resilience.

Result: We launched the feature on schedule across both platforms, achieved 95% call success rate, received positive user feedback (4.6 star rating), and the collaborative framework we established became the standard for future cross-platform features.

8. Describe a time when you had to advocate for technical improvements that weren't initially prioritized.

Situation: Our app had growing accessibility issues affecting users with disabilities, but leadership prioritized new features over accessibility improvements.

Task: I needed to build a compelling case for accessibility work and secure resources without appearing obstructive to business goals.

Action: I gathered data on our user base, researched legal requirements and market opportunities, created a presentation showing potential user reach (15% of users), demonstrated competitive advantages, and proposed a 20% time allocation model. I also built a prototype showing quick wins like proper content descriptions and TalkBack support.

Result: Leadership approved a dedicated accessibility sprint, we improved our accessibility score from 45 to 88, received recognition from accessibility advocacy groups, expanded our user base by 12%, and I became the accessibility champion for the organization.

9. Tell me about a challenging debugging experience and how you resolved it.

Situation: Users reported intermittent data loss in our note-taking app, but the issue was non-reproducible in our test environment and affected only 2% of users randomly.

Task: I needed to identify the root cause of this elusive bug that was damaging user trust and causing negative reviews.

Action: I implemented comprehensive logging with Firebase Analytics custom events, added local crash reporting for non-fatal exceptions, analyzed patterns in affected devices (discovered correlation with specific Android versions and OEMs), reproduced the issue using similar device configurations, and found a race condition in our Room database transactions during app backgrounding.

Result: I fixed the issue by implementing proper transaction handling and lifecycle-aware data persistence, data loss incidents dropped to zero, we recovered app rating from 3.8 to 4.5 stars, and implemented better monitoring to catch similar issues proactively.

10. Share an example of when you had to learn and implement a new Android technology quickly.

Situation: Google announced Jetpack Compose as the future of Android UI, and our company wanted to evaluate it for a greenfield project launching in 6 weeks.

Task: I was selected to learn Compose, assess its viability for our use case, and potentially lead the implementation despite having no prior Compose experience.

Action: I dedicated evenings to Google's Compose tutorials and codelabs, built a proof-of-concept implementing our core UI components, evaluated performance and interop with existing View-based code, documented learnings and trade-offs, and presented findings to the team. After approval, I created coding guidelines and conducted knowledge-sharing sessions.

Result: We successfully launched the project using Compose, reduced UI code by 30%, improved development speed for UI features by 40%, and I became the internal Compose expert, later training 15+ developers across the organization.