MASTER PREP — Android Developer Interview (incl. Jetpack Compose)

Mukesh Kumar Patel | One file. Everything you need. Interview: tomorrow 6 PM.

⭐ = highest-probability question. If short on time, do ⭐ only.

PART 1 — SELF INTRODUCTION

60–90 second version (memorize this)

> *"I'm an Android developer with 4.5+ years of experience building production apps in Kotlin. Currently I'm at Netlink Software, and before that I spent 2.5 years at ShopKirana working on B2B commerce apps.*
>
> *On the native side, I've built reactive UIs with Jetpack Compose, Material 3, and StateFlow, following MVVM and Clean Architecture with Dagger Hilt. I led a Java-to-Kotlin migration at ShopKirana that significantly cut null-pointer crashes, and I built offline-first ordering with Room so field sales agents could work without internet, with automatic sync on reconnect. I've also integrated real-time chat with Socket.IO, payments with Razorpay, and the full Firebase stack — FCM, Crashlytics, Remote Config.*
>
> *My apps have 50,000+ cumulative downloads on the Play Store, including a government app serving citizens across Madhya Pradesh, and I've personally handled versioning and staged rollouts on the Play Console.*
>
> *I also work with Flutter, but native Android with Kotlin and Compose is my core strength, and I enjoy owning features end-to-end — from architecture to release."*

30-second version (if they say "briefly")

> *"Android developer, 4.5+ years, Kotlin and Jetpack Compose. I've shipped production apps with 50K+ downloads — B2B commerce at ShopKirana where I led a Java-to-Kotlin migration and built offline-first ordering with Room, and a government services app. Strong in MVVM/Clean Architecture, Hilt, Coroutines, and Compose."*

Your 3 core stories (STAR — rehearse out loud)

Story 1 — SK Agent (offline-first + migration) — your strongest:

- **S:** Field sales agents had poor connectivity but needed to place orders and track deliveries.
- **T:** Maintain and enhance the app, migrate legacy Java to Kotlin, add offline-first ordering.
- **A:** Built offline cart with Room as single source of truth + sync queue on reconnect; led module-by-module Java→Kotlin migration prioritizing high-crash paths; added COD, Pay Later, and QR payment modes; FCM for order/delivery updates.
- **R:** Fewer NPE crashes post-migration, agents work fully offline, improved production stability.

Story 2 — Home-Workout (your Compose showcase):

- **S:** Wanted a fitness tracker that works fully offline and stays fluid during live workout tracking.
- **T:** Build native Android app from scratch with modern stack.
- **A:** 100% Kotlin + Jetpack Compose + Material 3; StateFlow + Coroutines for reactive UI; Room for workout history/stats; strict MVVM/Clean layers (Compose UI → ViewModel → Repository → Room DAO).
- **R:** Responsive UI during live tracking, zero network dependency, easily debuggable codebase.

Story 3 — MP eNagarpalika (scale + compliance):

- **S:** MP citizens needed digital access to municipal services (property tax, water tax, sewerage) with strict government compliance.
- **T:** Build a scalable app integrated with government backends.
- **A:** Clean Architecture with feature modules; REST integration with auth tokens, retries, response caching; extensive QA/backend collaboration for compliance.
- **R:** Statewide production app, met compliance and performance benchmarks, live on Play Store.

Defend your resume numbers (one line each)

- **40% less boilerplate** → Hilt replaced manual DI/factories/singletons across modules.
- **Fewer crashes post-migration** → null-safe Kotlin APIs; prioritized highest-crash paths first; verified in Crashlytics.
- **50K+ downloads** → cumulative across production apps including eNagarpalika.
- **25% responsiveness (Netlink)** → state-management refactor, fewer rebuilds, response caching, proper async error/retry handling.

Behavioral one-liners

- **Why leave current job?** "I'm looking for deeper native Android/Compose ownership and bigger technical challenges." (Positive only. Never badmouth.)
- **Conflict:** API design disagreement on eNagarpalika — resolved by prototyping both and comparing with data.
- **Failure:** A production bug that reached users — caught via Crashlytics, hotfixed, then added a regression test + staged rollout policy so it can't repeat.
- **Deadline:** Play Store release under time pressure — scoped ruthlessly, shipped core, follow-up release for the rest.

Questions YOU ask (pick 2–3)

1. "How much of the codebase is Compose vs XML today — is there an active migration?"
2. "What would success look like in my first 90 days?"
3. "How does the team ship — release cadence, CI/CD, staged rollouts?"

PART 2 — 150 QUESTIONS WITH DETAILED ANSWERS

SECTION A: JAVA (Q1–Q25)

Q1. ⭐ Difference between JDK, JRE, and JVM?

JVM (Java Virtual Machine) executes compiled bytecode. It's what makes Java platform-independent — each OS has its own JVM implementation, but they all run the same `.class` bytecode. It handles class loading, bytecode verification, JIT compilation, and garbage collection.

JRE (Java Runtime Environment) = JVM + core libraries (java.lang, java.util...). Enough to RUN Java programs, not to compile them.

JDK (Java Development Kit) = JRE + development tools (javac compiler, debugger, javadoc). What developers install.

Android note: Android doesn't use a standard JVM — it uses ART (Android Runtime), which compiles DEX bytecode ahead-of-time/JIT with profile-guided compilation.

Q2. Why is Java platform-independent?

Java source compiles to **bytecode** (`.class`), not machine code. Bytecode runs on any JVM regardless of OS/hardware — "write once, run anywhere." The JVM itself is platform-specific, but it presents a uniform execution environment. C/C++ by contrast compile directly to machine code per platform.

Q3. ⭐ Difference between `==` and `.equals()` ?

- `==` compares **references** for objects (do both variables point to the same memory?) and **values** for primitives.
- `.equals()` compares **logical equality** — but only if overridden (String, Integer override it; default Object.equals is same as `==`).

```
String a = new String("hi");
String b = new String("hi");
a == b        // false — different objects
a.equals(b)   // true — same characters
```

Trap they test: `Integer` caches -128..127, so `Integer a = 100; Integer b = 100; a == b` is true, but with 200 it's false. Always use equals for objects.

Q4. String vs StringBuilder vs StringBuffer?

- **String** — immutable. Every concat creates a NEW object. `s += "x"` in a loop = $O(n^2)$ garbage.
- **StringBuilder** — mutable char buffer, not thread-safe, fastest. Use for building strings in a method.
- **StringBuffer** — same API but synchronized methods; legacy, rarely needed.

Rule: loop concatenation → StringBuilder; simple one-line concat → String is fine (compiler optimizes it).

Q5. Why are Strings immutable in Java?

1. **String pool** — literals are shared; if mutable, changing one would change all references.
2. **Security** — file paths, URLs, class names passed as Strings can't be mutated after validation.
3. **Thread safety** — immutable = safely shareable across threads.
4. **Hashing** — String caches its hashCode; safe as HashMap keys (a mutable key would corrupt the map).

Q6. What are wrapper classes and autoboxing?

Object versions of primitives: `int→Integer`, `double→Double`, etc. Needed because collections/generics only hold objects. **Autoboxing** = automatic primitive→wrapper conversion; **unboxing** = reverse.

Pitfalls: unboxing a null Integer throws NPE; boxing in hot loops creates garbage (performance).

Q7. `final` vs `finally` vs `finalize()`?

- **final** — variable: can't reassign; method: can't override; class: can't extend (String is final).
- **finally** — block after try/catch that ALWAYS runs (cleanup: close streams). Runs even on return/exception; skipped only on System.exit or process kill.
- **finalize()** — method GC *might* call before reclaiming an object. Unreliable and deprecated — never use it; use try-with-resources instead.

Q8. Checked vs unchecked exceptions?

- **Checked** — compiler forces you to handle or declare (`IOException`, `SQLException`). Extend Exception.
- **Unchecked** — runtime, no forced handling (`NullPointerException`, `IllegalArgumentException`, `IndexOutOfBounds`). Extend RuntimeException.

Kotlin note (good to mention): Kotlin has NO checked exceptions — the compiler never forces try/catch. Cleaner interop but you must know your APIs.

Q9. Can you override a static method?

No. Static methods belong to the **class**, not instances, and are resolved at **compile time**. Declaring the same static method in a subclass is **method hiding**, not overriding — which one runs depends on the reference type, not the object type. No runtime polymorphism for statics.

Q10. What is a constructor? Can it be private?

Special method that initializes a new object; same name as class, no return type; chained with `this()` / `super()`.

Yes, private constructors are used for:

- **Singleton** — only the class itself can instantiate.
- **Utility classes** — prevent instantiation entirely.
- **Factory methods** — force creation through named static methods (`User.of(...)`).

Q11. ⭐ ArrayList vs LinkedList?

- **ArrayList** — backed by a resizable array. O(1) random access by index; insert/delete in the middle is O(n) (shifts elements); occasional resize+copy on growth.
- **LinkedList** — doubly-linked nodes. O(1) insert/delete at a known node, O(n) access by index; higher memory per element (node objects + pointers); poor cache locality.

In practice: **ArrayList wins almost always** on modern hardware due to cache locality. LinkedList only when doing many head/middle insertions via iterator, or as a Deque.

Q12. ⭐ HashMap vs Hashtable vs ConcurrentHashMap?

- **HashMap** — not thread-safe; allows one null key and null values; fastest single-threaded.
- **Hashtable** — legacy; every method synchronized (one big lock = slow); no nulls. Don't use.
- **ConcurrentHashMap** — thread-safe with fine-grained locking (per-bucket CAS + synchronized bins in Java 8), so readers never block and writers rarely contend. No null keys/values. The correct choice for concurrent access.

Q13. ⭐ How does HashMap work internally?

1. `put(key, value)`: compute `key.hashCode()`, spread it, map to a bucket index: `(n-1) & hash`.
2. Empty bucket → store new node. Occupied → compare with `equals()`; same key → replace value; different → **collision**: append to linked list in that bucket.
3. Java 8+: a bucket's list converts to a **red-black tree** after 8 nodes (O(n) → O(log n) worst case).
4. When size exceeds capacity × load factor (default 0.75), the table **resizes** (doubles) and entries are rehashed.
5. `get`: same hashing to find bucket, then equals() along list/tree.

Q14. The hashCode/equals contract?

- If `a.equals(b)` then `a.hashCode() == b.hashCode()` — MUST hold.
- Equal hashCodes do NOT imply equals (collisions are legal).
- Break the contract (override equals but not hashCode) → object goes into one HashMap bucket by identity hash, then lookup by an equal key computes a different bucket → **you can't find your own data**.

Kotlin `data class` generates both correctly — mention that.

Q15. HashSet vs TreeSet vs LinkedHashSet?

- **HashSet** — HashMap-backed; O(1) add/contains; no order.
- **LinkedHashSet** — insertion order preserved; slightly more memory.
- **TreeSet** — red-black tree; sorted order; O(log n); supports range queries (headSet, ceiling); elements must be Comparable or given a Comparator.

Q16. Comparable vs Comparator?

- **Comparable** — the class defines its own natural order: `class User implements Comparable<User> { compareTo(...) }`. One ordering per class.
- **Comparator** — external ordering object: `Comparator.comparing(User::getAge)`. Many orderings, no class modification, composable with `thenComparing`.

Kotlin: `sortedBy { it.age }` / `compareBy(...)` do the same, cleaner.

Q17. What are generics and why use them?

Compile-time type parameters: `List<String>` instead of raw `List`. Benefits: type safety (no ClassCastException at runtime), no manual casting, self-documenting APIs.

Type erasure: generics exist only at compile time; at runtime `List<String>` is just `List`. That's why you can't do `new T()` or `instanceof List<String>` — and why Kotlin's `reified` (with inline) is special: it keeps the real type at the call site.

Q18. Abstract class vs interface (Java 8+)?

- **Abstract class**: can have state (fields), constructors, any-visibility methods; a class extends only ONE.
- **Interface**: contract; default + static methods since Java 8; only public static final fields; a class implements MANY.

Choose abstract class for shared *implementation and state* among closely related classes; interface for a *capability* (Comparable, Clickable) across unrelated classes. Prefer interfaces — more flexible.

Q19. The `static` keyword? Static block?

Static members belong to the class, one copy shared by all instances, accessible without an instance. Static blocks run once when the class loads (init static state). *Android caution:* a `static` reference to an Activity/Context = classic memory leak, because class-level references outlive the Activity.

Q20. Is Java pass-by-value or pass-by-reference?

Always pass-by-value. For objects, the *value of the reference* is copied — so the method can mutate the object's fields (both references point to the same object) but CANNOT reassign the caller's variable.

```
void change(User u) { u.name = "X"; }   // affects caller's object
void swap(User u)   { u = new User(); } // caller unaffected
```

Q21. ⭐ Memory leaks in Java/Android — what and common causes?

A leak = objects no longer needed but still **reachable from a GC root**, so GC can't reclaim them. Android-specific classics:

1. **Static reference to Activity/Context/View** — outlives the Activity.
2. **Non-static inner classes / anonymous Handlers** — hold implicit `this` to Activity; a delayed message keeps the dead Activity alive.
3. **Unremoved listeners/callbacks** — registered to long-lived objects (system services, buses).
4. **Coroutines/Rx not cancelled** — running work referencing UI.
5. **Singletons caching views/bitmaps.**

Detection: **LeakCanary** in debug, Memory Profiler heap dumps in Studio. Fix: use viewModelScope/lifecycleScope, unregister in onDestroy/DisposableEffect, applicationContext where appropriate, WeakReference as last resort.

Q22. How does Garbage Collection work?

GC frees objects unreachable from **GC roots** (thread stacks, static fields, JNI refs). Modern collectors are **generational**: most objects die young, so heap splits into young generation (frequent, fast **minor GC**) and old generation (survivors promoted; **major GC** less often). ART on Android uses a concurrent copying collector to keep pauses short. You don't call GC manually — you just avoid leaks and excessive allocation (object churn in onDraw/composition causes jank via GC pressure).

Q23. `volatile` vs `synchronized`?

- **volatile** — guarantees **visibility**: reads/writes go to main memory, no thread-local caching, plus ordering (happens-before). Does NOT make compound actions atomic (`count++` is still broken).

- **synchronized** — mutual exclusion (one thread in the block) + visibility on lock boundaries. Use for read-modify-write.
- Also: `AtomicInteger` etc. for lock-free atomic ops.

Kotlin equivalents: `@Volatile`, `synchronized(lock){}`, or better — confine state to a single dispatcher/Mutex in coroutines.

Q24. What is a deadlock and how do you prevent it?

Thread A holds lock 1 and waits for lock 2; thread B holds lock 2 and waits for lock 1 — both wait forever. Prevention:

1. **Consistent lock ordering** (always acquire lock1 before lock2 everywhere).
2. Avoid nested locks; keep critical sections tiny.
3. Use timeouts (`tryLock`) instead of blocking forever.
4. Prefer higher-level constructs: coroutines + Mutex, single-threaded confinement, immutable data.

Q25. ⭐ Thread lifecycle and ways to create threads?

Lifecycle: **New → Runnable → Running → Blocked/Waiting/Timed-waiting → Terminated.**

Creation: extend `Thread`, implement `Runnable` (preferred — composition), `Callable` + Future (returns value), **ExecutorService** thread pools (production standard).

Android answer that scores points: "On Android I don't manage raw threads — I use **Kotlin coroutines** with Dispatchers.IO/Default, which pool threads underneath and add structured concurrency, cancellation, and lifecycle awareness."

SECTION B: OOPS (Q26–Q45)

Q26. ⭐ The 4 pillars of OOP?

1. **Encapsulation** — bundle data + methods, hide internals behind access modifiers. *Example: my Repository exposes `getWorkouts(): Flow<List<Workout>>`; whether it's Room or network is hidden.*
2. **Abstraction** — expose WHAT, hide HOW, via interfaces/abstract classes. *Example: ViewModel depends on a `WorkoutRepository` interface, not the Room implementation.*
3. **Inheritance** — IS-A reuse: subclass gets parent's behavior. *Example: `BaseViewModel` with shared error handling.*
4. **Polymorphism** — one interface, many implementations; calls resolve to the actual object's method at runtime. *Example: sealed `UiState` handled exhaustively in `when`.*

Q27. ⭐ Encapsulation vs Abstraction?

Both hide things, but different things:

- **Encapsulation hides DATA/state** — private fields, controlled access, protecting invariants. Implementation-level.
- **Abstraction hides IMPLEMENTATION/complexity** — the caller sees a simple contract. Design-level.

Concrete: `private val _uiState = MutableStateFlow(...)` exposed as `val uiState: StateFlow<...>` = encapsulation (UI can read, only ViewModel writes). The repository *interface* the ViewModel talks to = abstraction.

Q28. What is inheritance? Types?

Subclass inherits parent's members and can override behavior (IS-A). Types: **single** (one parent), **multilevel** (A→B→C), **hierarchical** (one parent, many children). Java/Kotlin forbid **multiple class inheritance** (diamond problem) but allow implementing multiple interfaces. Kotlin classes are `final` by default — you must mark them `open`, a deliberate design-for-inheritance choice.

Q29. ⭐ Compile-time vs runtime polymorphism?

- **Compile-time (overloading)** — same method name, different parameter lists; resolved by the compiler from argument types. Includes operator overloading in Kotlin.
- **Runtime (overriding)** — subclass redefines a parent's method; the JVM picks the implementation from the **actual object type** at runtime (virtual dispatch). This is "real" polymorphism enabling `List<Shape>` where each `draw()` behaves differently.

Q30. Rules for method overriding?

- Same name + parameter list; return type same or **covariant** (subtype).
- Cannot **reduce** visibility (public → protected is illegal); can widen.
- Cannot throw broader **checked** exceptions.
- `final` / `static` / `private` methods can't be overridden.
- Kotlin: parent method must be `open`, child must write `override` (explicitness prevents accidents).

Q31. What is the diamond problem?

Class D inherits from B and C, both of which override a method from A — which version does D get? Ambiguity. Java/Kotlin ban multiple class inheritance to avoid it. With **interfaces**, if two default/interface implementations clash, the compiler FORCES you to override and choose: in Kotlin, `super<B>.foo()` / `super<C>.foo()`.

Q32. Composition vs inheritance — preference?

"Favor composition over inheritance." Inheritance couples you to the parent's implementation (fragile base class problem), is fixed at compile time, and single-slot. Composition (HAS-A: inject collaborators) is flexible, swappable at runtime, and testable with fakes.

Your example: ViewModel doesn't EXTEND some data class — it's COMPOSED with a Repository injected by Hilt; tests pass a fake repository. Inheritance is right for genuine IS-A framework contracts (ViewModel(), RecyclerView.Adapter).

Q33. Coupling and cohesion?

- **Coupling** — how much modules depend on each other's internals. Want **LOW**: change one module without breaking others. Achieved via interfaces, DI, layering.
- **Cohesion** — how focused a module is on one responsibility. Want **HIGH**: everything in a class relates to one job.

Clean Architecture is exactly this: high cohesion within layers (domain does business logic only), low coupling between layers (interfaces + dependency rule).

Q34. ⭐ SOLID — explain with examples from your work.

- **S (Single Responsibility)** — one reason to change. *Use cases in my domain layer:* `PlaceOrderUseCase` *does exactly one thing.*
- **O (Open/Closed)** — open to extension, closed to modification. *Sealed* `UiState` + `when`*: adding a new state extends the hierarchy; compiler flags every* `when` *to update — no old code silently breaks.*
- **L (Liskov Substitution)** — subtype must work anywhere the parent is expected. *Any* `WorkoutRepository` *implementation (Room, fake, remote) must honor the same contract so ViewModel never knows the difference.*
- **I (Interface Segregation)** — many small interfaces > one fat one. *Separate* `OrderReader` / `OrderWriter` *rather than a 20-method god-interface.*
- **D (Dependency Inversion)** — depend on abstractions, not concretions; high-level policy doesn't import low-level detail. *ViewModel → repository INTERFACE; the Room implementation is wired by Hilt in the data layer.*

Q35. What is dependency injection and why?

Objects receive their dependencies from outside instead of constructing them. Benefits: **testability** (inject fakes/mocks), **decoupling** (swap implementations), **single configuration point**, lifecycle/scope management.

Your line: "I use Hilt — constructor injection into ViewModels and repositories. In tests I pass fake repositories directly. At ShopKirana, moving to Hilt cut ~40% of manual factory/singleton boilerplate."

Q36. Singleton — implementation and pitfalls?

Java: private constructor + static instance; thread-safe variants: eager init, double-checked locking with volatile, enum singleton. **Kotlin: just `object` — thread-safe by language guarantee.**

Pitfalls: global mutable state (hidden coupling), hard to test (can't substitute), lifetime = process (Android: never let a singleton hold Activity context — leak). Better: DI-scoped `@Singleton` via Hilt — same lifetime, but injectable and fakeable.

Q37. Factory pattern — where used?

Creation logic behind a method so callers don't `new` concrete classes: centralizes construction, enables polymorphic return. Android examples you actually touched: `ViewModelProvider.Factory` (constructing ViewModels with args pre-Hilt), `Retrofit.create(ApiService::class.java)`, `Fragment.instantiate`. Kotlin's companion-object factory methods (`User.from(json)`) are the idiomatic light version.

Q38. Builder pattern — why and where in Android?

Constructs complex objects step by step with readable, optional parameters instead of telescoping constructors. Android: `AlertDialog.Builder`, `NotificationCompat.Builder`, `Retrofit.Builder`, `OkHttpClient.Builder`, `Room.databaseBuilder`. *Kotlin note:* named + default arguments replace most hand-written builders; `apply {}` gives builder-like syntax for free.

Q39. Observer pattern — where in Android?

A subject maintains observers and notifies them of changes — decouples producer from consumers. Everywhere in Android: **LiveData.observe**, **Flow collection**, click listeners, `BroadcastReceiver`, `ContentObserver`, Room's invalidation tracker (DB change → Flow re-emits), and Compose recomposition itself (composables "observe" the state they read).

Q40. Repository pattern — explain from YOUR apps.

Single source of truth that abstracts WHERE data comes from. My flow: **ViewModel → Repository interface → (Room DAO | Retrofit API)**. The repository decides: serve cache, hit network, or merge. In SK Agent, the repository wrote orders to Room immediately (offline) and synced to the server when connectivity returned — the ViewModel never knew or cared. Benefits: swap/fake for tests, centralized caching and error mapping, UI totally decoupled from data sources.

Q41. Adapter pattern in Android?

Converts one interface into another the client expects. Textbook Android case: **RecyclerView.Adapter** adapts your `List<Item>` into ViewHolders the RecyclerView can render. Also Retrofit's `CallAdapter` (Call → suspend/Flow) and Moshi/Gson `TypeAdapter`s.

Q42. What is an anti-pattern? One you've fixed?

A common solution that looks fine but causes long-term damage. Ones I've fixed: **God Activity** (1000+ lines of UI + business + network logic) — during the ShopKirana migration I extracted ViewModels, use cases, and repositories, which is what made features unit-testable. Others: hardcoded strings/dimens, callback hell (→ coroutines), doing DB/network on the main thread, treating fragments as reusable Views.

Q43. Interface vs abstract class — how to choose?

Ask: am I sharing a **contract** or an **implementation**?

- Capability that unrelated classes can have → **interface** (Clickable, Comparable).
- Common state + partial implementation for a close family → **abstract class** (BaseViewModel with shared error channel).
- Need multiple? Only interfaces. Kotlin interfaces can hold default method bodies but no state — the practical dividing line is *fields/constructors*.

Q44. What is immutability and why does it matter?

Object state cannot change after construction (`val` fields, no setters, copies for change). Wins: **thread safety** without locks, predictable code (no action-at-a-distance), safe HashMap keys, and critically for you — **Compose stability**: immutable data classes let Compose skip recomposition reliably. Pattern: `data class` + `val` + `copy()` for modified versions; `List` (read-only) over `MutableList` in exposed APIs.

Q45. Association vs Aggregation vs Composition?

Increasing strength of "has" relationships:

- **Association** — uses / knows about: `Doctor` treats `Patient`; independent lifecycles.
- **Aggregation** — has, but parts survive the whole: `Team` has `Players`; disband the team, players exist.
- **Composition** — owns, dies together: `Activity` and its `Views`; destroy the Activity, its view tree is gone.

SECTION C: KOTLIN (Q46–Q80)

Q46. ⭐ `val` vs `var` vs `const val`?

- **val** — read-only reference, assigned once at runtime. NOT deep immutability: `val list = mutableListOf(1)` — reference fixed, contents mutable.
- **var** — mutable reference, reassignable.

- **const val** — **compile-time** constant: value inlined at call sites at compile time; only primitives/String; only top-level or inside `object`/companion.

Default to `val` everywhere; use `var` only when reassignment is genuinely needed — fewer moving parts, Compose-friendlier.

Q47. ⭐ How does Kotlin null safety work?

Nullability is part of the **type system**: `String` can never be null, `String?` can. The compiler forces you to handle the null case before use:

- `?.` safe call — `user?.name` → null if user is null (chainable).
- `?:` elvis — `user?.name ?: "Guest"` → default when null.
- `!!` not-null assertion — "crash here if null" (NPE). Avoid; every `!!` is a code smell.
- **Smart cast** — after `if (x != null)`, compiler treats x as non-null in that scope.
- `let` — `user?.let { ... }` runs block only when non-null.

Your hook: "This is exactly why the Java→Kotlin migration at ShopKirana cut our NPE crashes — nullability moved from runtime surprises to compile-time errors."

Q48. ⭐ `lateinit` vs `lazy`?

- **lateinit var** — "I'll initialize before first use, I promise." Non-null `var`, no primitives, for framework-injected things (views, DI fields). Access before init → `UninitializedPropertyAccessException`. Check with `::field.isInitialized`.
- **val by lazy { ... }** — computed **once on first access**, then cached. It's a `val`, works with any type, thread-safe by default (SYNCHRONIZED mode). For expensive objects you may never need.

Rule: lateinit = externally initialized `var`; lazy = self-initializing `val`.

Q49. ⭐ What does a `data class` generate?

`equals()`/`hashCode()` (from primary-constructor properties — contract kept automatically), `toString()`, `copy()` (clone with selective changes — key for immutable state updates: `state.copy(loading = false)`), and `componentN()` for destructuring (`val (id, name) = user`).

Restrictions: primary constructor needs ≥1 val/var param; can't be abstract/open/sealed/inner. *Compose tie-in:* immutable data classes (all `val`) are STABLE → Compose can skip recomposition.

Q50. ⭐ What is a `sealed class` / `sealed interface`?

Restricted hierarchy — all subclasses known at compile time (same module). Superpower: **exhaustive** `when` — the compiler errors if you miss a case, no `else` needed. Add a new subclass → every `when` in the codebase flags itself for update. That's Open/Closed principle enforced by the compiler.

```
sealed interface UiState {
    object Loading : UiState
    data class Success(val data: List<Workout>) : UiState
    data class Error(val message: String) : UiState
}
```

Your hook: "Every screen in my apps models UI state exactly like this — the ViewModel emits it via StateFlow, Compose renders the `when`."

Sealed vs enum: enum = fixed set of constant instances; sealed = fixed set of TYPES, each carrying different data.

Q51. `object` vs `companion object`?

- **object** — declaration = thread-safe lazy singleton in one keyword. Also anonymous objects (`object : ClickListener {...}`).
- **companion object** — a singleton *attached to a class*: Java-static-like members (`User.create()`, constants, factory methods). Can implement interfaces; `@JvmStatic` for Java interop.

Q52. ⭐ Scope functions — let, run, with, apply, also?

Two axes: context object as **it** or **this**; returns **lambda result** or **the object**.

Function	Object as	Returns	Use for
`let`	it	lambda result	null-safe transform: `user?.let { save(it) }`
`run`	this	lambda result	configure + compute a result
`with`	this	lambda result	group calls on an object (non-null)
`apply`	this	**the object**	object configuration: `Intent().apply { putExtra(...) }`
`also`	it	**the object**	side effects in a chain: `.also { log(it) }`

Don't over-nest them — readability first. Interviewers like hearing that.

Q53. Extension functions — power and limitations?

Add functions to types you don't own: `fun String.isValidEmail(): Boolean`. Compiles to a static function taking the receiver as first parameter — so: resolved **statically** (no polymorphism — extension chosen by declared type, not runtime type), can't access private members, member functions always win over same-signature extensions.

Great for: utility APIs (`Context.toast(...)`), keeping data classes clean, adapting third-party types. Compose's entire Modifier system is extension functions.

Q54. `==` vs `===` in Kotlin?

- `==` → **structural** equality, compiles to `equals()` (null-safe: `null == null` is true).
- `===` → **referential** — same instance.

So Kotlin fixed Java's trap: `==` on Strings does what beginners expect.

Q55. Default and named arguments?

Defaults kill overload explosion: one function replaces four Java overloads. Named args make call sites self-documenting and let you skip middle parameters:

```
fun fetch(page: Int = 1, size: Int = 20, refresh: Boolean = false)
fetch(refresh = true)
```

Compose relies on this completely — every composable has dozens of defaulted params.

Q56. `when` vs `switch`?

`when` is an **expression** (returns a value), no fallthrough/break, matches ranges (`in 1..10`), types (`is String`), arbitrary conditions (no argument form), multiple values per branch — and is **exhaustive** with sealed classes/enums (compiler-checked). It's pattern matching light, not just a switch.

Q57. Higher-order functions and lambdas?

Functions that take or return functions; function types like `(Int) -> String` are first-class. Lambda syntax: `{ x -> x * 2 }`, single param as `it`, trailing lambda outside parens — which is why Compose looks like a DSL: `Column { Text("hi") }` is just a trailing lambda. This + extension receivers = Kotlin DSLs.

Q58. What does `inline` do?

Compiler copies the function body AND its lambda arguments directly into the call site — no Function object allocation, no virtual call. Use for small, frequently-called higher-order functions (the stdlib's `let/map/forEach` are inline). Bonus: inline lambdas allow **non-local return** and enable **reified** type parameters. Don't inline big functions — code bloat.

Q59. `noinline` and `crossinline`?

Within an inline function:

- **noinline** — exclude one lambda from inlining so it can be stored in a field or passed elsewhere (an inlined lambda has no object to store).

- **crossinline** — lambda will be called from another context (nested lambda/other thread), so forbid non-local `return` inside it while still inlining.

Q60. String templates and raw strings?

`"$name scored ${list.size}"` — expressions inline, no concatenation noise. Raw strings `"""..."""` — multi-line, no escaping (regex, JSON, SQL for Room migrations), with `.trimIndent()` for clean formatting.

Q61. ⭐ How is Kotlin null safety better than Java — YOUR migration story?

Java: any reference can be null; you find out at **runtime** as an NPE in production. Kotlin: nullability is in the type; the **compiler** blocks unhandled nulls before you ship.

Full story to tell: "At ShopKirana I led the Java→Kotlin migration of SK Agent. We went module by module, prioritizing paths with the highest crash counts in Crashlytics. Converting forced us to decide, for every field, is this actually nullable? — which surfaced dozens of latent NPE paths. Combined with moving to Kotlin ViewModels + Hilt, our crash-free session rate measurably improved. Interop annotations (@Nullable/@NonNull) on the remaining Java kept the boundary safe during the transition."

Q62. Why `?.let` over `if (x != null)` sometimes?

Smart cast doesn't work on mutable state: a `var` or a class property could be changed by another thread between the check and the use, so the compiler refuses to smart-cast it. `x?.let { }` captures the value once, so it's safe. For local `val`s, plain if-null-check is fine and often more readable.

Q63. ⭐ What is a `suspend` function — and under the hood?

A function that can **pause without blocking its thread** and resume later; callable only from a coroutine or another suspend function. At a suspension point the thread is released to run other work.

Under the hood: the compiler performs **CPS transformation** — adds a hidden `Continuation` parameter and compiles the body into a **state machine**, where each suspension point is a state. "Resuming" = invoking the continuation with the result. Cheap: a suspended coroutine is a small object, not a blocked thread — you can run 100K coroutines where 100K threads would kill the app.

Q64. ⭐ `launch` vs `async`?

- **launch** → returns `Job`; fire-and-forget; uncaught exception propagates immediately up the scope (crash unless handled).
- **async** → returns `Deferred<T>`; produces a value via `await()`; exception is stored and thrown **at await()**.

Use async for **parallel decomposition**:

```
val profile = async { api.getProfile() }
val orders  = async { api.getOrders() }
show(profile.await(), orders.await())   // both ran in parallel
```

Never use async as fire-and-forget — swallowed exceptions.

Q65. ⭐ Dispatchers — Main, IO, Default, Unconfined?

- **Main** — Android UI thread: UI updates, collecting flows for UI. `Main.immediate` avoids re-dispatch when already on main.
- **IO** — blocking I/O: network, disk, DB. Large elastic pool (64+) since threads mostly wait.
- **Default** — CPU-bound work: JSON parsing, sorting, image processing. Pool sized to CPU cores.
- **Unconfined** — no thread confinement; resumes wherever; basically never in app code.

Best practice you should say: **inject dispatchers** (constructor param) so tests can pass a TestDispatcher; Room/Retrofit already switch to their own executors, so don't wrap every call in withContext(IO) blindly.

Q66. ⭐ What is structured concurrency?

Every coroutine lives inside a **scope**, forming a parent-child tree:

1. Parent **waits** for all children before completing.
2. Cancelling parent **cancels all children**.
3. A child's failure cancels the parent and siblings (unless SupervisorJob).

No orphaned background work, no leaks. *Android reality:* `viewModelScope` — everything launched there is auto-cancelled in `onCleared()`; `lifecycleScope` + `repeatOnLifecycle(STARTED)` for UI collection that stops when backgrounded.

Q67. `CoroutineScope` vs `CoroutineContext`?

- **Context** — an immutable set of elements: `Job` (lifecycle), `Dispatcher` (threading), `CoroutineName`, `CoroutineExceptionHandler`. Combined with `+`: `Dispatchers.IO + SupervisorJob()`.
- **Scope** — a holder of one context that defines the **lifetime boundary**; you call `launch` / `async` ON a scope; children inherit its context.

Q68. How does cancellation work?

Cooperative — cancellation sets the Job's state, but running code must notice: all `kotlinx.coroutines` suspend functions (delay, withContext) check and throw `CancellationException`. A CPU-bound loop with no suspension points won't stop — add `ensureActive()` or `yield()` inside loops. CancellationException is *normal* — never

swallow it in a generic catch (rethrow it). Cleanup that must survive cancellation: `withContext(NonCancellable) { }`.

Q69. `Job` vs `SupervisorJob`?

- **Job** — a child's failure cancels the parent and all siblings (fail together).
- **SupervisorJob** — children fail **independently**; one crashed child doesn't kill the rest.

`viewModelScope` = `SupervisorJob() + Dispatchers.Main.immediate` — one failed screen operation shouldn't kill every other coroutine of the ViewModel. `supervisorScope { }` for a local block with the same behavior.

Q70. Exception handling in coroutines?

1. **try/catch around suspend calls** — normal, preferred, keeps errors local (my repositories catch and map to a domain Result/sealed error).
2. **CoroutineExceptionHandler** — last-resort handler for uncaught exceptions in `launch` (install at scope root: logging/Crashlytics).
3. **async** — exception surfaces at `await()`; wrap the await.
4. `supervisorScope` to isolate sibling failures.
5. Never catch-and-ignore `CancellationException`.

Pattern to quote: repository returns `Result<T>` or sealed `NetworkResult`, ViewModel maps it to `UiState.Error(message)` — exceptions never cross layer boundaries raw.

Q71. ⭐ What is Flow? Cold vs hot?

`Flow<T>` = asynchronous stream of values built on coroutines.

- **Cold (Flow)** — code runs **per collector**, starts on `collect`, like a recipe. Room DAO flows are cold: each collector gets its own query + updates.
- **Hot (StateFlow/SharedFlow)** — active regardless of collectors; emissions shared; late collectors miss (or replay) earlier values, like a live broadcast.

Backpressure is natural: emission suspends until the collector is ready.

Q72. ⭐ StateFlow vs SharedFlow vs LiveData?

- **StateFlow** — hot; **always has a value** (initial required); conflates (skips intermediate values, keeps latest); replays current value to new collectors; `distinctUntilChanged` built in. → **UI STATE**.
- **SharedFlow** — hot; configurable `replay`/buffer; no initial value; every emission delivered. → **ONE-TIME EVENTS** (navigation, snackbar) with replay=0.
- **LiveData** — lifecycle-aware, main-thread, Android-only; fine in legacy View code.

Why StateFlow in Compose: Kotlin-first, no lifecycle owner needed, full operator set, testable without InstantTaskExecutorRule; collect via `collectAsStateWithLifecycle()`. *Your line:* "StateFlow for state, SharedFlow for events — state can be re-rendered any time, an event must fire exactly once; putting a navigation event in StateFlow re-navigates on rotation."

Q73. ⭐ map vs flatMapLatest vs collectLatest?

- **map** — transform each value 1:1.
- **flatMapLatest** — each value starts a NEW inner flow and **cancels the previous one**. Classic: search — new query cancels the in-flight request: `queryFlow.debounce(300).flatMapLatest { repo.search(it) }`.
- **collectLatest** — terminal: new emission **cancels the still-running block** for the previous one — UI only cares about latest.

Siblings: flatMapConcat (sequential), flatMapMerge (concurrent). "Latest" variants = cancellation-based freshness.

Q74. What does `flowOn` do?

Changes the dispatcher of the **upstream only** (operators/emission above it); downstream and collection stay on the collector's context. `flow { heavyRead() }.flowOn(Dispatchers.IO)` → produce on IO, collect on Main. Context preservation rule: a flow can't emit from a different dispatcher directly — flowOn is the sanctioned mechanism.

Q75. stateIn / shareIn — why?

Convert a **cold** flow into a **hot** shared one so N collectors don't trigger N upstream executions (N DB queries, N network calls).

```
val workouts = repo.observeWorkouts()
    .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())
```

`WhileSubscribed(5000)` — upstream stops 5s after last collector leaves (survives rotation without restarting, stops when app is backgrounded). That 5000 is THE canonical Android answer — say it.

Q76. What is `reified` and why does it need `inline`?

Generics are erased at runtime (Q17) — normally you can't do `T::class`. In an **inline** function the body is copied into each call site with the **actual concrete type substituted**, so the type survives:

```
inline fun <reified T> Gson.fromJson(json: String): T =
    fromJson(json, T::class.java)
val user: User = gson.fromJson(json)   // no class token passed
```

Used all over Android: `Intent(context, T::class.java)`, Koin/Moshi/Nav APIs.

Q77. Delegated properties?

`val x by delegate` — get/set forwarded to the delegate's getValue/setValue. Stdlib: **lazy** (Q48), **observable** (callback on change), **map-backed**. Android: `by viewModels()`, `by dataStore(...)`. Custom example worth quoting — a SharedPreferences delegate:

```
var token: String by PrefsDelegate("auth_token", "")
```

Class delegation too: `class Tracker(analytics: Analytics) : Analytics by analytics` — composition with zero boilerplate.

Q78. What is a `value class`?

`@JvmInline value class UserId(val value: String)` — a type-safe wrapper that the compiler **erases at runtime** (passes the underlying String, no allocation in most positions). Fixes "stringly-typed" bugs: can't pass an OrderId where a UserId is expected, at zero cost. Boxing happens when used as a generic/nullable — know that caveat.

Q79. `Sequence` vs `List` operations?

List operators are **eager** — every `map/filter` allocates a full intermediate list. Sequence is **lazy** — elements flow one at a time through the whole chain, and short-circuit terminals (`first { }`) stop early.

Rule: small collections / 1-2 operators → List (simpler, often faster). Large collections + long chains or early termination → `asSequence()`. (Flow is the ASYNC lazy stream; Sequence is sync.)

Q80. Operator overloading — and in Compose?

Predefined operator functions: `plus` (+), `get` ([]), `invoke` (()), `contains` (in), compareTo... Kotlin allows only the fixed operator set — no arbitrary symbols, so it stays readable. In Android/Compose: `16.dp` (extension property), `Offset + Offset`, `padding(horizontal = 8.dp)`, destructuring via componentN, and `invoke` powering function-like objects.

SECTION D: ANDROID (Q81–Q115)

Q81. ⭐ Explain the Activity lifecycle.

onCreate (create UI, restore state, init ViewModel) → **onStart** (visible, not interactive) → **onResume** (foreground, interactive) → user leaves → **onPause** (losing focus — keep it FAST, blocks next app) → **onStop** (not visible —

release heavier resources) → **onDestroy** (finishing or config change) — or back via **onRestart** → onStart.

Nuances that show seniority: onPause is guaranteed, onDestroy is NOT (process kill); multi-window means another app being focused only pauses you; register/unregister symmetrically (onStart/onStop pairs).

Q82. ⭐ What happens on rotation? Why does ViewModel survive?

Rotation = **configuration change**: by default the Activity is destroyed and recreated (onPause→onStop→onDestroy→onCreate→onStart→onResume) so resources can be re-resolved (layout-land, etc.).

ViewModel survives because it isn't stored in the Activity — it lives in the **ViewModelStore**, which the system retains across recreation (via NonConfigurationInstances). The new Activity instance asks `ViewModelProvider` for the same key and gets the **same instance**. `onCleared()` is called only when the Activity is *finishing* for real.

That's why: state in ViewModel + StateFlow → rotation is a non-event; UI just re-subscribes and re-renders.

Q83. ⭐ Config change vs process death?

- **Config change** — Activity recreated, **process lives**: ViewModel survives, statics survive.
- **Process death** — system kills your backgrounded app for memory: ViewModel GONE, memory GONE. User returns → system restores the task and hands back the **saved instance state Bundle**.

Handling: **SavedStateHandle** in ViewModel (survives BOTH — it's Bundle-backed), `rememberSaveable` in Compose, onSaveInstanceState in Views. Rule: UI state (scroll, text input, selected id) → saved state; heavy data → reload from Room/network on restore. Test it: Developer options → "Don't keep activities", or `adb shell am kill <package>`.

Q84. Fragment lifecycle — key extras?

Adds: **onAttach** (context available) → onCreate → **onCreateView** (inflate) → **onViewCreated** (view setup here) → ... → **onDestroyView** (VIEW dies, fragment object may live — back stack!) → onDestroy → onDetach.

The classic bug: observing LiveData/Flow with `this` (fragment) as lifecycle owner instead of **viewLifecycleOwner** → duplicate observers or leaked view references after back-stack navigation. Also null-out view bindings in onDestroyView. (Compose makes this whole class of bugs disappear — worth saying.)

Q85. Intent — implicit vs explicit?

Intent = messaging object to start Activities/Services or deliver broadcasts, carrying extras (Bundle).

- **Explicit** — target class named: `Intent(this, DetailActivity::class.java)` — your own screens.
- **Implicit** — declare action/data/category (`ACTION_VIEW` + url); system finds matching components via **intent filters**; multiple matches → chooser. Android 11+ package visibility rules apply to querying handlers.

Q86. Activity launch modes?

- **standard** — new instance every launch (default).
- **singleTop** — if already **on top**, reuse it → `onNewIntent()` instead of new instance (notifications tapping into the current screen).
- **singleTask** — one instance per task; relaunch **clears activities above it** and calls onNewIntent (app entry points, deep-link roots).
- **singleInstance** — singleTask + it's ALONE in its task (rare: call screens).

Also via flags: `FLAG_ACTIVITY_NEW_TASK | CLEAR_TOP`. In single-Activity Compose apps this mostly moves to the NavController — good observation to make.

Q87. Service types — started vs bound vs foreground?

- **Started** (startService) — runs until stopSelf/stopService; background execution limits (Android 8+) kill background-started services quickly.
- **Bound** (bindService) — clients bind via ServiceConnection/Binder; lives while anyone is bound; for in-process client-server (music player controls).
- **Foreground** — startForeground + **persistent notification**; user-visible ongoing work: media, navigation, **my GPS tracking app** — location FGS with the `location` foregroundServiceType, started only during active tracking.

Android 8+: background apps must use `startForegroundService` and promote within ~5s, or use WorkManager instead.

Q88. ⭐ WorkManager vs Foreground Service vs AlarmManager?

- **WorkManager** — **deferrable, guaranteed** work: survives process death and reboot, respects Doze, supports constraints (network, charging), backoff/retry, chaining. My use: **offline order sync on reconnect** (network constraint) in SK Agent.
- **Foreground Service** — work the user is actively aware of, RIGHT NOW, continuous: tracking, playback.
- **AlarmManager** — wall-clock exact triggers (alarm at 6:00); `setExactAndAllowWhileIdle` + exact-alarm permission (12+).

Decision: user-visible-now → FGS; must-happen-eventually → WorkManager; exact time → AlarmManager. FCM push for server-initiated wakeups.

Q89. BroadcastReceiver — restrictions on 8+?

Responds to system/app events (connectivity, boot, battery). Android 8+ removed most **implicit broadcasts from the manifest** (apps waking en masse killed batteries) — exceptions: BOOT_COMPLETED, LOCALE_CHANGED, etc. Alternatives: **context-registered receivers** (register/unregister with lifecycle — only while relevant), WorkManager

constraints instead of CONNECTIVITY_CHANGE, JobScheduler. Android 13+: exported/not-exported flag required for context-registered receivers.

Q90. What is a ContentProvider?

Standard interface for sharing structured data **across apps** via `content://` URIs with query/insert/update/delete — contacts, MediaStore, FileProvider (sharing files safely via content URIs instead of file:// which is banned since N). Also the trick behind library auto-init (androidx.startup uses a provider because providers initialize before Application.onCreate). Within one app, Room + DI is the normal answer — you don't need a provider for yourself.

Q91. ⭐ What is ANR?

Application Not Responding dialog — main thread blocked too long: ~**5s** for input dispatch, 10s broadcasts, 20s foreground services. Causes: network/DB on main thread, heavy JSON parsing, synchronous IPC, lock contention, deadlocks.

Prevention: everything heavy off main via **coroutines on IO/Default**; Room forbids main-thread queries by default (say this); StrictMode in debug; monitor **Play Console vitals + ANR traces** — ANR rate affects Play visibility. My apps: repositories suspend on injected IO dispatcher; main thread only renders state.

Q92. View measure/layout/draw?

Three passes down the view tree: **onMeasure** (parent gives MeasureSpecs, child reports size) → **onLayout** (parent positions children) → **onDraw** (Canvas rendering). `invalidate()` = redraw only; `requestLayout()` = re-measure + re-layout. Deep nesting multiplies measure passes (why flat ConstraintLayout beat nested LinearLayouts). Compose's equivalent: composition → layout → draw, with the rule that a Compose layout measures children **exactly once** — a big perf win.

Q93. ⭐ Explain MVVM in YOUR apps.

- **View** (Compose/Activity) — renders state, forwards user events. Dumb.
- **ViewModel** — holds UI state (`StateFlow<UiState>`), handles events, calls domain/data layer; survives config change; **no Context, no View references**.
- **Model** — repositories, use cases, data sources.

My concrete pattern: single `UiState` data class (or sealed class) per screen; ViewModel exposes `StateFlow` via `stateIn(WhileSubscribed(5000))` ; Compose collects with `collectAsStateWithLifecycle` ; events flow UP as function calls (`viewModel::onAddClicked`), state flows DOWN — **unidirectional data flow**. Result at ShopKirana: ViewModels fully unit-testable (fake repository + TestDispatcher, assert state transitions).

Q94. ⭐ Clean Architecture layers + dependency rule?

- **Presentation** — Compose UI + ViewModels. Knows domain.

- **Domain** — entities + use cases + repository **interfaces**. Pure Kotlin, ZERO Android imports. Knows nobody.
- **Data** — repository **implementations**, Room, Retrofit, DTOs + mappers. Implements domain interfaces.

Dependency rule: source dependencies point INWARD (presentation → domain ← data). Inner layers never know outer ones. Wins: domain testable as plain JUnit, swap data sources without touching UI, team scaling via clear boundaries. Honest senior note: for small apps I've merged use-case layers where they'd be pass-through — architecture serves the app, not dogma. My flow: `Compose → ViewModel → UseCase → Repository interface → (Room | Retrofit)`.

Q95. ⭐ How does Hilt work? Key annotations?

Compile-time DI built on Dagger — generates the object graph as code (fast at runtime, errors at build time, not runtime).

- `@HiltAndroidApp` on Application — generates the root component.
- `@AndroidEntryPoint` on Activity/Fragment/Service — enables field injection there.
- `@HiltViewModel` + `@Inject constructor` — ViewModels with dependencies, integrated with `by viewModels()` / `hiltViewModel()`.
- `@Module` + `@InstallIn(SingletonComponent::class)` + `@Provides` (builder-created things: Retrofit, Room) / `@Binds` (interface → implementation).
- Scopes: `@Singleton`, `@ViewModelScoped`, `@ActivityScoped` — instance lifetime tied to component lifetime.
- Qualifiers (`@Named`/custom) to disambiguate same-type bindings (two OkHttpClients).

Your line: "At ShopKirana, Hilt replaced hand-written singletons/factories — about 40% less boilerplate and constructor-injected ViewModels made unit testing trivial."

Q96. Constructor vs field injection?

Constructor injection: dependencies **explicit and final** — the class doesn't compile without them; tests just call the constructor with fakes (no DI framework in unit tests at all). Field injection: hidden deps, mutable, order-of-init hazards — only for framework classes you don't construct (Activities), which is exactly where Hilt uses it. Rule: constructor injection everywhere you own the constructor.

Q97. ⭐ Room — components and migrations?

- `@Entity` (table), `@Dao` (interface: @Query/@Insert/@Update/@Delete, suspend or Flow returns), `@Database` (holder, version, exportSchema).
- Also: `@TypeConverter` (Date↔Long), `@Transaction` (atomic multi-op), `@Relation` / `@Embedded` for relations.

Migration — schema change requires version bump + migration or Room throws:

```
val M_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE workouts ADD COLUMN notes TEXT")
    }
}
```

`fallbackToDestructiveMigration()` **wipes data** — dev only, never production (my apps carry user carts/history — data loss unacceptable). Test with `MigrationTestHelper` + exported schema JSONs in version control. autoMigrations handle simple cases with annotations.

Q98. Room + Flow?

DAO returns `Flow<List<T>>` → Room's **invalidation tracker** watches the touched tables and **re-runs the query and re-emits automatically on any change**. UI observes DB → insert from anywhere (sync worker, another screen) → UI updates itself. This is the reactive backbone of offline-first: my workout history screen and SK Agent cart both render straight from Room flows — no manual refresh anywhere.

Q99. ⭐ Your offline-first design (SK Agent)?

1. **Room = single source of truth** — UI ALWAYS renders from DB flows, never from network responses directly.
2. **Reads**: network fetch → upsert into Room → Flow re-emits → UI updates. Offline → cached data still renders.
3. **Writes offline**: order saved to Room with `PENDING_SYNC` status + queued.
4. **Sync**: WorkManager job with network constraint drains the queue on reconnect (retry + backoff); status flips to `SYNCED` (or `FAILED` for user attention).
5. **Conflicts**: business rule per entity — server-wins for catalog/prices, client-preserved for the agent's cart; server validated final order state.
6. UI shows sync state honestly (pending badge) — agents trusted the app because it never silently lost an order.

Q100. ⭐ Retrofit — auth tokens and 401 handling?

Two mechanisms:

- **Interceptor** — appends `Authorization: Bearer <token>` to every request (reads from encrypted storage).
- **okhttp3.Authenticator** — invoked by OkHttp **on 401**: refresh the token (synchronously, mutex-guarded so N failed requests don't fire N refreshes), return the original request rebuilt with the new token → OkHttp retries automatically. Refresh fails → return null + emit logout event.

I built this pattern at HackerKernel with interceptors and refined it later. Also: attach interceptor order matters (auth before logging), and never log tokens.

Q101. Application vs network interceptor (OkHttp)?

- **Application interceptor** (`addInterceptor`) — runs ONCE per call, sees your original request and final response, unaware of redirects/retries/cache — right place for auth headers, logging your API contract.
- **Network interceptor** (`addNetworkInterceptor`) — runs per actual network hop, sees wire-level request (after redirects, real headers), skipped entirely on cache hits — for wire debugging, response rewriting.

Rule of thumb: 95% of the time you want application interceptors.

Q102. FCM end-to-end?

1. App start → FCM SDK issues a **registration token** (`onNewToken` — send to your backend, tokens rotate!).
2. Server sends via FCM HTTP v1 API targeting token/topic.
3. Two payload types — THE key detail:
   - **Notification message** — app in background: system posts the notification itself, `onMessageReceived` NOT called (tap → intent extras).
   - **Data message** — `onMessageReceived` always called (app alive) → you build the notification, deep-link, update Room.
4. Priority: high wakes device through Doze (use sparingly), normal is batched.

My use: order/delivery status updates in SK Agent, promotional pushes in Zila — data messages so we control channels, deep links, and quiet hours.

Q103. PendingIntent and immutability flags?

A token that grants another process (system, notification manager, AlarmManager) permission to fire YOUR Intent later with YOUR identity. Since Android 12, must declare `FLAG_IMMUTABLE` (receiver can't modify the wrapped intent — security) or `FLAG_MUTABLE` (only when the system must fill things in, e.g. notification direct replies). Also know: request codes + FLAG_UPDATE_CURRENT to update the extras of an existing PendingIntent.

Q104. SharedPreferences vs DataStore?

SharedPreferences: synchronous API tempting main-thread IO, `apply()` async-but-unsignaled / `commit()` blocking, no change notifications, can lose data on process kill mid-write.

DataStore: coroutine/Flow-based (reads are a Flow — reactive settings!), transactional `edit {}`, guaranteed consistency, Proto DataStore adds full type safety via schema. Migration path from SP is built in. New code → DataStore; my answer for why: "settings become just another Flow my ViewModel combines into UiState."

Q105. How do you secure an Android app?

- **Storage**: EncryptedSharedPreferences / Jetpack Security, keys in **Android Keystore** (hardware-backed), never plaintext tokens.

- **Network**: HTTPS only, **certificate pinning** (OkHttp CertificatePinner) for high-risk apps — I'd cite payment flows; Network Security Config to forbid cleartext.
- **Code**: R8 obfuscation; NO secrets/API keys hardcoded (extractable from APK) — keys behind backend or NDK at minimum.
- **Payments (my Razorpay work)**: order created server-side, signature verification server-side — client never trusted with amounts.
- **Other**: FLAG_SECURE on sensitive screens, exported=false by default, validate deep-link inputs, ProGuard rules audited.

Q106. R8/ProGuard — what and keep rules?

R8 (default since AGP 3.4): **shrinks** (removes unused code — tree shaking), **obfuscates** (renames to a/b/c), **optimizes** (inlining, dead branch removal), plus resource shrinking. Breaks anything found via **reflection** — Gson/Moshi model classes, JNI, XML-referenced classes → `-keep` rules (`-keep class com.app.data.model.** { *; }`) or `@Keep`. Libraries ship consumer rules. Always test the **release** build — the classic "works in debug, crashes in release" is a missing keep rule; verify with mapping.txt + re-trace for Crashlytics.

Q107. ⭐ Finding and fixing a memory leak?

Process: (1) **LeakCanary** in debug builds — automatic heap dump + leak trace on retained Activities/Fragments; (2) read the trace: GC root → chain of references → leaked object; the fix target is the link that SHOULDN'T exist; (3) Android Studio **Memory Profiler** for manual heap dumps and allocation tracking; (4) confirm fixed by repeating the leak scenario.

Common fixes: unregister listeners symmetrically, viewModelScope instead of GlobalScope, no static Activity/View refs, applicationContext for long-lived objects, null view bindings in onDestroyView, DisposableEffect cleanup in Compose. Signals in prod: rising OOM crashes in Crashlytics, memory vitals.

Q108. Why can't a ViewModel hold Activity context?

ViewModel **outlives** the Activity across config changes — rotation destroys the Activity, but the ViewModel keeps its reference → the ENTIRE view hierarchy can't be GC'd → leak per rotation. If context is genuinely needed (string resources, system services): `@ApplicationContext` injected via Hilt (process-lifetime, safe) — or better, resolve strings in the UI layer and keep the ViewModel Android-free for testability.

Q109. ⭐ Battery-optimized GPS tracking (your project)?

From my HackerKernel tracker:

1. **FusedLocationProviderClient** (GPS+WiFi+cell fusion), not raw GPS.
2. **Right priority**: PRIORITY_BALANCED_POWER_ACCURACY when moving slowly, HIGH_ACCURACY only when needed — biggest single battery lever.
3. **Adaptive intervals** + `setMinUpdateDistanceMeters` — no updates when stationary.

4. **Batching** (setMaxWaitTime) — deliver several fixes in one wakeup.
5. **Foreground service only during active tracking** — started/stopped explicitly, never all-day background polling.
6. Batched network uploads of points, not per-fix requests. Result: accuracy the client needed at acceptable battery cost.

Q110. Doze mode and App Standby?

Doze — device stationary + screen off → system defers network, jobs, syncs, regular alarms into periodic **maintenance windows**; deepens over time. **App Standby** — per-app: unused apps get restricted buckets (rare/restricted) with tighter job/alarm quotas.

Survive it correctly: WorkManager (respects windows), **high-priority FCM** for genuinely urgent server pushes (punches through Doze), setExactAndAllowWhileIdle for real alarms. Don't ask users for battery-optimization exemption unless the app's core purpose demands it (Play policy).

Q111. Background location on Android 10+?

Android 10 split out **ACCESS_BACKGROUND_LOCATION**; 11+ you can't even request it in the normal dialog — user must pick "Allow all the time" in **Settings** (system sends them there via your rationale flow). Foreground-only alternative: a **foreground service** with `foregroundServiceType="location"` counts as foreground access — this is how my tracker worked: request while-in-use permission, run an FGS during active tracking sessions. Play Console requires a declaration + video for background-location apps. Incremental ask (foreground first, background later with rationale) is the approved UX.

Q112. Reducing app startup time?

Measure first: **Macrobenchmark** + Perfetto, cold/warm/hot distinction, TTID/TTFD.

1. **Baseline Profiles** — pre-compile hot paths; typically 20-30% cold-start win, biggest cheap lever.
2. Slim **Application.onCreate** — no synchronous SDK inits; androidx.**App Startup** or lazy-init-on-first-use; move non-critical init post-first-frame.
3. Defer heavy DI graphs (Hilt is compile-time — cheap already), avoid main-thread disk reads (StrictMode catches).
4. Simple first frame: splash screen API, avoid overdraw/deep hierarchies (Compose helps).
5. R8 + resource shrinking — smaller dex loads faster.

Q113. App Bundle vs APK; staged rollouts (you did this)?

AAB — publishing format; Play generates **split APKs** per device (ABI, density, language) → users download only what their device needs (~20-30% smaller). Required for new apps since 2021. Dynamic feature modules for on-demand delivery.

Staged rollout (my Play Console workflow): release to 5-10% → watch **Crashlytics + Android vitals** (crash rate, ANR rate) for a day or two → expand 25→50→100%, or **halt** on regression; fix, bump version, resume. In-app updates API (flexible/immediate) to move stragglers off broken versions. Also: release notes, pre-launch reports, managed publishing for coordinated launches.

Q114. Gradle build — variants, flavors?

Build types (debug/release: minify, signing, debuggable) × **product flavors** (free/paid, dev/staging/prod: applicationId suffix, BuildConfig fields, per-flavor resources) = **variants** (prodRelease...). My uses: `buildConfigField` for per-environment BASE_URL, separate Firebase configs per flavor, signingConfigs for release. Modern hygiene worth mentioning: version catalogs (libs.versions.toml), KSP over kapt, configuration cache, convention plugins in multi-module builds.

Q115. Multi-module architecture — why?

1. **Build speed** — Gradle rebuilds only changed modules, parallelizes, caches; the big win at scale.
2. **Enforced boundaries** — :domain literally cannot depend on :app; the dependency rule becomes a compile error, not a convention.
3. Ownership/team scaling; 4. Dynamic feature delivery; 5. Reuse across apps.

Typical shape: `:app` (wiring/nav) → `:feature:*` → `:core:domain` / `:core:data` / `:core:ui` / `:core:common`. Honest cost note: module boilerplate + build-logic complexity — worth it as team/app grows, overkill for a 3-screen app.

SECTION E: JETPACK COMPOSE (Q116–Q150)

Q116. ⭐ What is Compose and how is it different from XML Views?

Declarative UI toolkit in pure Kotlin: **UI = f(state)** — you describe what the UI looks like for a given state; when state changes, the framework re-executes affected composables and updates only what changed. You never mutate a view tree.

vs XML/Views: no findViewById/ViewBinding, no view-state synchronization bugs (the #1 source of UI bugs — updating a view and forgetting another), single language for UI+logic, much less code, previews, built-in animation APIs. Views are imperative retained-mode objects; Compose regenerates descriptions and diffs. *Your hook:* "My Home-Workout app is 100% Compose + Material 3 — after years of XML, the elimination of state-sync bugs is the biggest practical win."

Q117. ⭐ What is a composable function?

`@Composable fun` — a function that **emits UI** into the composition. Key properties:

- Can only be called from other composables (compiler-enforced context).
- **Restartable** — the runtime can re-invoke it independently (recomposition).
- **Skippable** — if inputs haven't changed (and are stable), the runtime skips it.
- Should be **idempotent and side-effect free** — same inputs → same UI, no mutations during execution (side effects go in effect handlers, Q128+).
- Can run in any order and on multiple threads (future-proofing) — another reason to avoid side effects.

Q118. ⭐ What is recomposition? What triggers it?

Recomposition = re-executing composables whose **observed state changed**, to update the UI. Compose's **snapshot state system** records which composables READ which `State` objects during composition; a WRITE to that state invalidates exactly those readers — scoped, not whole-tree.

"Smart recomposition": within an invalidated scope, child composables whose parameters are **unchanged and stable** are skipped. Triggers: `mutableStateOf` writes, StateFlow emissions via collectAsState, animation frames. NOT triggered by plain variable changes — state must be Compose-observable.

Q119. ⭐ remember vs rememberSaveable?

- **remember { }** — caches a value in the composition, surviving **recomposition** (otherwise every recomposition would reset it). Dies with the composable leaving composition, and on config change.
- **rememberSaveable { }** — additionally survives **configuration change and process death** by saving into the instance-state Bundle. Bundle-able types automatic; custom types need a **Saver**.

Rule: transient UI state (expanded flag, text input) → rememberSaveable; anything important → ViewModel + SavedStateHandle. remember(key1, key2) — recompute when keys change.

Q120. ⭐ mutableStateOf — how does Compose track it?

`MutableState<T>` is an **observable snapshot state holder**. During composition, every `.value` READ is recorded against the current recompose scope; a WRITE looks up subscribed readers and schedules exactly those for recomposition. Built on the **snapshot system** — MVCC-like isolation that also makes state thread-safe to observe.

```
var count by remember { mutableStateOf(0) }   // property delegate syntax
```

Always pair with remember inside composables (else reset every recomposition). Also know: `mutableStateListOf` / `mutableStateMapOf` for observable collections — mutating a plain `MutableList` inside state does NOT trigger recomposition (classic bug).

Q121. ⭐ State hoisting — what and why?

Move state UP out of a composable; the child receives `value: T` + `onValueChange: (T) -> Unit` and becomes **stateless**:

```
@Composable fun SearchBar(query: String, onQueryChange: (String) -> Unit)
```

Why: **single source of truth** (no duplicated/desynced state), **reusability** (component doesn't own policy), **testability** (pass state, assert UI), state accessible to siblings above. Hoist state to the **lowest common ancestor** that needs it — screen-level state to the ViewModel. This is the mechanical core of **unidirectional data flow**: state down, events up.

Q122. Stateful vs stateless composables?

- **Stateless** — everything via parameters; pure render function; the default for reusable/library components.
- **Stateful** — owns `remember` state internally; convenient for self-contained details no caller cares about (e.g. a ripple, an internal animation).

Common pattern: provide both — a stateful overload wrapping a stateless core — callers pick their level of control. Material components follow this.

Q123. Column vs Row vs Box?

- **Column** — vertical sequence; `verticalArrangement` (spacing/distribution) + `horizontalAlignment`.
- **Row** — horizontal; `horizontalArrangement` + `verticalAlignment`.
- **Box** — stacks children (z-order = declaration order); `contentAlignment`, per-child `Modifier.align`; for overlays, badges, backgrounds.
- Plus `Modifier.weight` in Row/Column for proportional space — the LinearLayout-weight equivalent.

Q124. Modifier — and does order matter?

Ordered chain of decorations: size, padding, background, clickable, clip... Each wraps the next — so **order absolutely matters**:

```
Modifier.padding(16.dp).background(Red)   // red INSIDE padding
Modifier.background(Red).padding(16.dp)   // red includes padding area
```

Same with `clickable` before/after padding (does the padding area ripple?). Also: Modifier is a parameter convention — every reusable composable should accept `modifier: Modifier = Modifier` and apply it to its root; that's an API-design point interviewers listen for.

Q125. LazyColumn vs Column + scroll?

- **LazyColumn/LazyRow** — composes **only visible items** (+ a small buffer), recycles compositions as you scroll; DSL: `items(list, key = {...}) { }`. The RecyclerView successor — no Adapter/ViewHolder/DiffUtil boilerplate.
- **Column + verticalScroll** — composes **ALL children immediately**; fine for a small fixed set (a settings screen), disastrous for hundreds of items.

Also in the family: LazyVerticalGrid, sticky headers, `rememberLazyListState` for scroll position and scroll-driven UI.

Q126. Clicks and ripple?

`Modifier.clickable(onClick = ...)` — Material ripple by default inside a themed hierarchy. Customization: `interactionSource` + `indication` (or `LocalIndication`); disable via `enabled=false`; `combinedClickable` for long-press; `pointerInput { detectTapGestures/detectDragGestures }` for raw gestures. Button/Card etc. have onClick built in.

Q127. Slot API pattern?

Composable lambda parameters as "slots" the caller fills with arbitrary UI:

```
Scaffold(topBar = { AppBar() }, floatingActionButton = { Fab() }) { padding ->
Content(Modifier.padding(padding)) }
```

Instead of a component exposing 30 config params, it exposes slots — maximally flexible, no inheritance needed. Material components are built this way (Button's content is a RowScope slot). Your reusable components should use it too — e.g. a generic `AppCard(header = {...}, body = {...})`.

Q128. ⭐ LaunchedEffect — what and when?

Runs a **suspend block in a coroutine scoped to the composition**: starts when it enters composition, **cancels and restarts when a key changes**, cancels on leave.

```
LaunchedEffect(userId) { viewModel.load(userId) }     // reload when id changes
LaunchedEffect(Unit) { snackbarHostState.showSnackbar(msg) }  // once per entering
```

Uses: one-shot loads keyed to inputs, collecting one-time event flows, animations on appear, debounced reactions to state. Key choice IS the semantics: `Unit` = once per composition lifetime; a value = restart-on-change.

Q129. ⭐ DisposableEffect — what and when?

For effects that need **cleanup**. Block runs on enter/key-change and MUST end with `onDispose { }`, which runs on leave or before re-run:

```
DisposableEffect(lifecycleOwner) {
    val observer = LifecycleEventObserver { _, e -> ... }
    lifecycleOwner.lifecycle.addObserver(observer)
    onDispose { lifecycleOwner.lifecycle.removeObserver(observer) }
}
```

Uses: lifecycle observers, broadcast receivers, sensor/location listeners, any register/unregister pair. This is Compose's structural answer to the forgot-to-unregister leak.

Q130. ⭐ rememberCoroutineScope vs LaunchedEffect?

Both give composition-scoped coroutines; the difference is **who initiates**:

- **LaunchedEffect** — *composition*-driven: runs because the composable appeared or a key changed.
- **rememberCoroutineScope** — *event*-driven: you get a scope to launch from **callbacks** (you cannot call a suspend function or LaunchedEffect inside onClick):

```
val scope = rememberCoroutineScope()
Button(onClick = { scope.launch { sheetState.show() } })
```

Rule: "should happen because UI is showing" → LaunchedEffect; "should happen because user acted" → rememberCoroutineScope.

Q131. ⭐ derivedStateOf — when?

For state **computed from other state that changes less often than its inputs**:

```
val showFab by remember { derivedStateOf { listState.firstVisibleItemIndex > 0 } }
```

Scroll index changes every frame while scrolling — but `showFab` only flips at the 0↔1 boundary. Readers recompose **only when the RESULT changes**. Without it: recomposition every scroll frame. Don't use when inputs and output change together (pure overhead) — that's simple calculation or `remember(key)`.

Q132. SideEffect and produceState?

- **SideEffect { }** — runs after every **successful** recomposition; for publishing Compose state to NON-Compose code (analytics, updating a legacy View or system UI controller).

- **produceState(initial, keys) { value = ... }** — converts non-Compose sources (callbacks, suspend calls, listeners) into `State<T>`; coroutine-scoped, keys restart it. Essentially LaunchedEffect + mutableStateOf fused.

Family summary you can rattle off: LaunchedEffect (suspend on enter/keys), DisposableEffect (cleanup), SideEffect (every recomposition), rememberCoroutineScope (callbacks), derivedStateOf (computed), produceState (external → State), rememberUpdatedState (capture latest value in long-lived effect).

Q133. ⭐ Collecting Flows in Compose properly?

`collectAsStateWithLifecycle()` (lifecycle-runtime-compose) — collection **stops when lifecycle drops below STARTED** (app backgrounded) and resumes on return. Plain `collectAsState()` keeps collecting in background → wasted work, and upstream (network/location) stays alive.

Pairs with `stateIn(WhileSubscribed(5000))` in the ViewModel: backgrounded → collectors gone → upstream stops after 5s; rotation's brief gap doesn't restart it. That pair of APIs is the complete modern answer — deliver it as one breath.

Q134. ⭐ Why StateFlow (not LiveData) with Compose?

1. Compose-natural: `collectAsStateWithLifecycle()` — no LifecycleOwner ceremony in composables.
2. **Always has a value** (initial required) — UI can render immediately, no nullable-first-emission dance.
3. Full Flow **operator set** (combine, map, debounce) for building UiState from multiple sources.
4. Kotlin-first, multiplatform, no main-thread setValue/postValue confusion; unit tests need no InstantTaskExecutorRule.
5. LiveData still fine in legacy View screens — but new Compose code has no reason for it.

Your hook: Home-Workout is exactly this — ViewModel StateFlow of UiState, Compose collects, `when` on the sealed state.

Q135. ⭐ One-time events (navigate/snackbar) in Compose?

The trap: put "navigate" in StateFlow/state → it **replays** on rotation/resubscribe → navigate again. Options:

1. **SharedFlow(replay=0) / Channel** in ViewModel, collected in a `LaunchedEffect`:

```
LaunchedEffect(Unit) { viewModel.events.collect { event -> when(event){...} } }
```

(Channel buffers while no collector — safer against dropped events.) 2. **Events-as-state** (Google's current guidance): event is a field in UiState; UI handles it then calls `viewModel.eventConsumed()` to clear. More boilerplate, fully state-driven and replay-proof.

Knowing BOTH and the trade-off is the senior answer.

Q136. CompositionLocal?

Implicit, tree-scoped values flowing down the composition without parameter-passing: `MaterialTheme` internals, `LocalContext`, `LocalDensity`. Provide via `CompositionLocalProvider(LocalSpacing provides Spacing()) { }`. `compositionLocalOf` (granular invalidation) vs `staticCompositionLocalOf` (rarely-changing, cheaper). Use ONLY for genuine ambient concerns (theming, locale) — using it to smuggle repositories/ViewModels hides dependencies and hurts testability. Explicit params > implicit locals by default.

Q137. Navigation in Compose?

`NavHost(navController, startDestination)` with `composable(route)` destinations; navigate via `navController.navigate(route)`; back stack managed for you.

- **Arguments**: route placeholders `"detail/{id}"` + NavType — or **type-safe routes** (Navigation 2.8+: @Serializable route classes — mention this, it's current).
- ViewModels scoped per destination via `hiltViewModel()`; SavedStateHandle receives nav args directly (my preferred pattern — ViewModel loads from args, UI stays dumb).
- Nested graphs for flows (auth vs main); `popUpTo(...) { inclusive }` / `launchSingleTop` for back-stack hygiene; pass **IDs, not objects**, in routes.

Q138. Theming with Material 3?

`MaterialTheme(colorScheme, typography, shapes) { content }` — components read these via CompositionLocals. Dark theme: `if (isSystemInDarkTheme()) darkColorScheme() else lightColorScheme()`. **Dynamic color** (Android 12+): `dynamicLightColorScheme(context)` derives the palette from the user's wallpaper. Semantic roles (primary, surfaceVariant, onPrimary...) — use roles, never hardcoded colors, and dark theme works for free. Extend with custom design tokens via your own CompositionLocals when the design system outgrows Material slots.

Q139. Animations in Compose?

Laddered toolbox — name them low→high:

- **animate*AsState** — one-liner value animation: `val size by animateDpAsState(target)`.
- **AnimatedVisibility** — enter/exit transitions (fade/slide/expand combos).
- **animateContentSize** — modifier; animates size changes automatically.
- **updateTransition** — multiple values coordinated by one state.
- **Animatable** + coroutines — gesture-driven, interruptible, spring physics.
- **AnimatedContent** — animating between content states; `animateItemPlacement` in lazy lists. All interruptible and state-driven by design — no XML animators, no end-listeners.

Q140. Interop — Compose in Views, Views in Compose?

- **ComposeView** in XML/Activity: `composeView.setContent { }` — how incremental migration starts (one screen, one card at a time).
- **AndroidView(factory = { MapView(it) }, update = { view -> ... })** inside Compose — for legacy/SDK views with no Compose equivalent (maps, ads, WebView).
- Themes bridged (Material theme adapters), shared ViewModels work across both worlds.

Your hook: "Same playbook as my Java→Kotlin migration — incremental, both worlds coexisting, high-change screens first, never big-bang."

Q141. Testing composables?

```
@get:Rule val rule = createComposeRule()
rule.setContent { WorkoutCard(workout, onClick = fake) }
rule.onNodeWithText("Push-ups").assertIsDisplayed()
rule.onNodeWithTag("start_button").performClick()
```

Finders (onNodeWithText/Tag/ContentDescription) work on the **semantics tree** (same tree accessibility uses — testTag or proper contentDescription = testable AND accessible). Assertions + actions (performClick, performTextInput); `waitUntil` for async. Stateless hoisted composables = trivially testable with direct state injection. Screenshot tests (Paparazzi/Roborazzi) for visual regressions — worth naming.

Q142. ⭐ Stability — what makes a type stable and why care?

A composable **skips** recomposition only if Compose can PROVE its parameters didn't change — requiring parameter types to be **stable**: equals is meaningful and Compose gets notified of changes (or values never change).

- Stable automatically: primitives, String, lambdas of stable captures, immutable data classes of stable vals, MutableState itself.
- **Unstable**: `List/Map/Set` interfaces (compiler can't prove the impl is immutable!), classes with `var`, classes from non-Compose modules without annotations.
- Fixes: `@Immutable` / `@Stable` annotations (promises to the compiler), kotlinx.collections.immutable (`ImmutableList`), or the **strong skipping mode** (Kotlin 2.0 compiler) which relaxes much of this — mentioning strong skipping marks you as current.

Diagnose with compiler metrics reports (which classes/params are unstable) — that detail lands very well.

Q143. ⭐ Optimizing LazyColumn performance?

1. `key = { it.id }` — stable identity across reorders/inserts: preserves item state and scroll position, avoids recomposing shifted items. #1 answer.
2. **contentType** for heterogeneous lists — recycling within type buckets.
3. **Immutable item models** (stable) → unchanged items skip.
4. Don't capture unstable lambdas per item — hoist callbacks (`onItemClick(id)` from parent, method references).
5. No unbounded nested lazies; fixed heights where possible; `derivedStateOf` for scroll-derived UI (Q131).
6. Measure: release build + Layout Inspector recomposition counts + Macrobenchmark scroll jank tests. Baseline Profiles help list startup jank.

Q144. ⭐ Why do lambdas cause recomposition and the fix?

A lambda that **captures unstable values or** `this` compiles to a new/unequal instance across recompositions → child's parameter "changed" → child never skips, cascading down entire lists.

Fixes:

1. **Method references to stable receivers**: `onClick = viewModel::onAddClicked` — stable.
2. Capture stable locals only; hoist the id: `{ onItemClick(item.id) }` where onItemClick comes stable from above.
3. `remember(keys) { lambda }` when capture is unavoidable.
4. Strong skipping mode (Kotlin 2.0) auto-memoizes lambdas — the modern mitigation, still worth understanding the mechanism.

Q145. "Defer state reads" optimization?

Pass a **lambda** instead of a value, so the state READ happens in the layout/draw phase, not composition:

```
Modifier.offset(x = scrollOffset)                // read in composition → recompose per frame
Modifier.offset { IntOffset(scrollOffset.toInt(), 0) }  // read in layout → NO recomposition
Modifier.graphicsLayer { alpha = fade.value }    // read in draw
```

Compose invalidates only the phase where the read happened — a parallax/scroll animation can run entirely in layout/draw at 60fps with **zero recompositions**. This answer separates senior Compose devs from juniors.

Q146. The three phases — Composition, Layout, Draw?

1. **Composition** — run composables → build/update the UI tree (what).
2. **Layout** — measure and place each node; single-pass: children measured **exactly once** (vs View's multi-pass) — why deep Compose trees don't explode like nested LinearLayouts.

3. **Draw** — render to canvas.

State-read phase determines invalidation scope (Q145): composition reads → recompose; layout reads (offset{}) → re-layout only; draw reads (graphicsLayer{}) → redraw only. Cheapest phase that can do the job wins.

Q147. @Immutable vs @Stable?

Both are **unchecked promises** to the compiler enabling skipping:

- **@Immutable** — "values NEVER change after construction" (stronger). For genuinely frozen models: `@Immutable data class WorkoutUi(...)` with vals of immutable types.
- **@Stable** — "may change, but equals is reliable AND Compose will be notified (mutations via snapshot state)". For observable holders: class with `var x by mutableStateOf(...)`.

Lie in the annotation → skipped recompositions with stale data — bugs. Prefer types that are provably stable; annotate at module boundaries where the compiler can't infer.

Q148. movableContentOf?

Wraps composable content so its **state survives moving to a different position in the tree**:

```
val player = remember { movableContentOf { VideoPlayer(state) } }
// use inside Row in landscape, Column in portrait — playback state preserved
```

Without it, moving = leave composition + re-enter = all remembered state reset. Uses: layout switches (orientation/adaptive), reparenting shared elements, reordering with heavy per-item state. Niche but a strong "knows the runtime" signal.

Q149. Debugging excessive recompositions?

1. **Layout Inspector** — live recomposition counts (blue) + skip counts per composable: find the hot ones.
2. **Composition tracing** — recomposition in Perfetto traces with composable names.
3. **Compiler metrics reports** (`-P ...reportsDestination`) — per-composable skippability + per-class stability: tells you WHY something never skips.
4. Temporary log/border in suspect composables for quick iteration.

Then fix with the toolkit: stability fixes (Q142), lambda memoization (Q144), derivedStateOf (Q131), deferred reads (Q145), lazy keys (Q143). Always measure on **release builds** — debug Compose is dramatically slower.

Q150. ⭐ Migrating a large XML app to Compose?

Incremental, never big-bang — both worlds coexist indefinitely:

1. Enable Compose in the build; bridge the theme (Material theme adapter) so components match.
2. **New screens/features in Compose first** — value immediately, zero rewrite risk.
3. Existing screens **bottom-up**: leaf components → ComposeView islands inside XML → whole screen when tipping point hits. AndroidView for the irreplaceables (maps/ads).
4. Architecture unchanged: same ViewModels/StateFlow serve both — migrating the VIEW layer only (if legacy screens are god-activities, extracting ViewModels comes first).
5. Prioritize **high-change screens** (recoup productivity), leave frozen screens last.
6. Team enablement + Compose-only rule for new UI, per-screen tracking.

Close with: "It's the same discipline as my Java→Kotlin migration at ShopKirana — incremental, crash-data-driven prioritization, both worlds stable throughout. That migration is how I know this playbook works."

PART 3 — HOW TO USE THIS FILE

Tonight (active recall, out loud — not silent re-reading):

1. Pass 1: ⭐ questions only (~45). Read the question, answer OUT LOUD, then check. Mark fumbles.
2. Pass 2: fumbled ones only.
3. Say the 60-second intro 3 times. Sleep by ~12:30. Sleep > one more hour of cramming.

Tomorrow:

- Commute (15 min): skim ⭐ questions, Compose section first (Q116–150).
- Lunch (20 min): fumbled questions + the 3 STAR stories.
- 5:00–5:45 PM: intro out loud twice, mic/camera check, water, silence phone. NO new topics.

In the interview:

1. Pause 3–5 seconds before answering. Structured and short beats long and rambling.
2. Formula: direct answer → brief why → "for example, in my SK Agent / Home-Workout app…"
3. Don't know? "I haven't used that in production, but my understanding is…" — never bluff.
4. Think out loud in coding rounds; state assumptions; test with an example.
5. Never badmouth employers. End with your 2–3 questions for them.