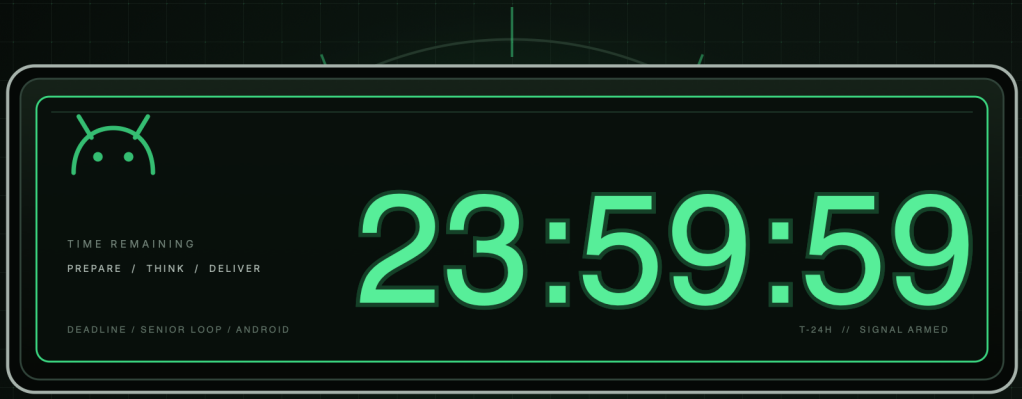
A | 24-HOUR INTERVIEW SYSTEM | 2026



THE 24-HOUR ANDROID INTERVIEW ANSWER BOOK

2026 SENIOR ANDROID QUESTIONS

Answers, follow-ups, traps, and recovery lines

plus live interview simulations under pressure

160 QUESTIONS | 14 VISUAL MODELS | 24-HOUR PLAN

MIKE SALARI

SALARI.DEV

Sample edition

Release date: August 2026

This sample contains the full contents of the complete book and three complete questions with their answer cards.

It intentionally excludes the live simulations, the 24-hour and 2-hour plans, the morning-of checklist, and the final spoken-answer drill. Those remain part of the complete edition.

The three cards show the format across Kotlin, coroutines, and Compose without turning the sample into a substitute for the book.

Table of contents from the complete book

This is the table of contents from the complete 428-page edition. It is included here so you can see the full scope of the book. It is not the contents of this sample.

The complete edition contains 160 answer cards and 14 original visual models.

Three questions, three complete answer cards

Q: In a large legacy codebase, you frequently see the non-null assertion operator. What problems does this create at scale, and how would you systematically reduce its usage without introducing runtime crashes?

Must-know · Very common · Mid to senior

SHORT ANSWER

Every `!!` is an unchecked nullability assertion that can turn a contract mistake into a production crash. I would inventory the sites, classify why each value is nullable, repair owned Java annotations and initialization boundaries first, then ratchet the remaining count downward in CI.

MEMORIZE THIS

Fix nullability at the boundary, then ratchet `!!` usage downward.

SENIOR ANSWER

Problems: each `!!` Is a latent KotlinNullPointerException with no context, they cluster in Java-interop and lateinit-adjacent code, and they train reviewers to ignore null contracts. Systematic reduction: inventory via detekt or lint counts per module; classify each site (truly impossible null, missing annotation on Java side, design gap); fix by class: add @Nullable and @NonNull annotations to owned Java code so the compiler sees platform types correctly, convert to safe call plus fallback where a default is meaningful, restructure initialization (constructor injection, lateinit with lifecycle guarantees, lazy) where the null window is temporal; ratchet with a CI count that must not grow.

WHY THEY ASK

This question reveals whether the candidate treats nullability as an API-contract problem or merely performs a syntax migration at individual call sites.

THE TRAP

The candidate mechanically replaces `!!` with `?.`, silently swallowing states that should fail loudly, while unannotated Java continues creating new platform types.

CODE

```
// Before: platform type forced !! everywhere.
val name = legacyUserService.getUser(id)!!.name!!

// After: annotate Java once, then handle the boundary.
val name = legacyUserService.getUser(id)?.name
    ?: error("User $id disappeared between list and detail")
```

FOLLOW-UP QUESTIONS

- When is an explicit failure preferable to a safe call with a default?
- How would you stop an unannotated Java boundary from reintroducing platform types?

LIVE INTERVIEW WORDING

I would not start with a global search-and-replace. First I would separate bad Java contracts, temporary initialization gaps, and values that are impossible to be null, because each category needs a different fix. Then I would enforce a declining module-level count so the migration cannot regress.

WEAK ANSWER

“I would replace every `!!` with `?.` and provide an empty default.”

INTERVIEWER PUSH

What happens when null represents corrupted state and your empty default lets the transaction continue?

RECOVERY LINE

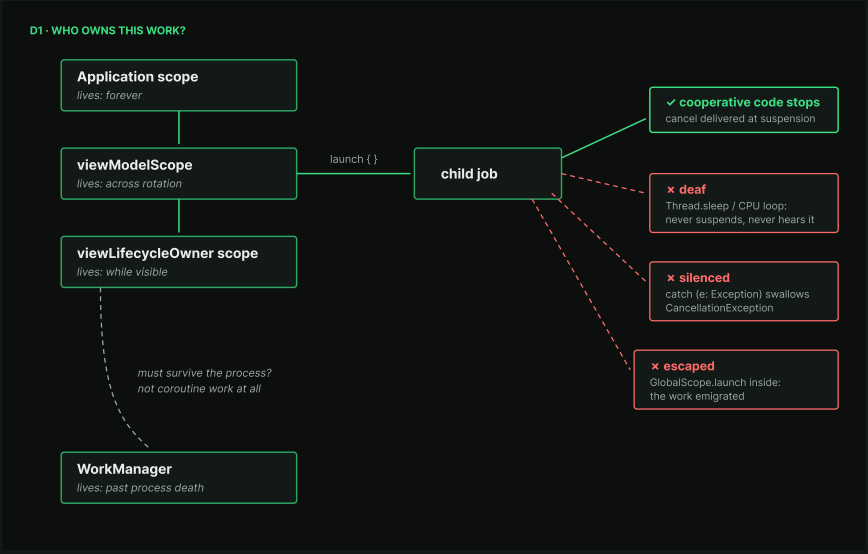
"I should classify the null contract first, repair the producing boundary, and use an explicit failure when continuing would hide invalid state."

WHAT EARNS THE POINT

The candidate identifies the producing boundary, distinguishes recoverable absence from invariant failure, and proposes a measurable ratchet rather than a blind replacement campaign.

Q: A coroutine launched in `viewModelScope` is still running after the ViewModel is cleared. What went wrong and how do you fix it?

Senior-depth · Common · Senior



Coroutine ownership: scope lifetime, cancellation, and the escape routes that cause leaks.

SHORT ANSWER

A correctly launched `viewModelScope` coroutine is cancelled when the ViewModel is cleared. If work survives, I look for a detached scope, blocking code that ignores cancellation, or work deliberately handed to a longer-lived owner such as `WorkManager`.

MEMORIZE THIS

Surviving work means ownership escaped or cancellation could not be observed.

SENIOR ANSWER

A `viewModelScope` job cancels on `onCleared`, but cancellation is cooperative. Blocking calls (Thread.sleep, blocking I/O, tight CPU loops) never observe cancellation. Other escapes: work handed to `GlobalScope` or an external scope inside the coroutine, NonCancellable contexts, or catching CancellationException and swallowing it. Fix: use suspending cancellation-aware APIs, check isActive or call ensureActive in loops, wrap blocking calls in withContext(Dispatchers.IO) knowing they still block until return, and rethrow CancellationException.

WHY THEY ASK

This question reveals whether the candidate can trace coroutine lifetime from the scope owner instead of treating `viewModelScope` as a cancellation guarantee for arbitrary work.

THE TRAP

The candidate adds another cancellation call in `onCleared()` while the real job was launched in `GlobalScope` or trapped inside non-cooperative blocking work.

CODE

```
class ReportViewModel(
    private val repo: ReportRepo
) : ViewModel() {
    fun generate() {
        viewModelScope.launch {
            val rows = repo.loadRows()
            val result = withContext(Dispatchers.Default) {
                rows.map { row ->
                    // Cooperates with ViewModel cancellation.
                    ensureActive()
                    heavyTransform(row)
                }
            }
            repo.save(result)
        }
    }
}
```

FOLLOW-UP QUESTIONS

- How would you prove which scope owns the surviving job?
- When should work outlive a ViewModel, and which component should own it?

LIVE INTERVIEW WORDING

I would first capture the job hierarchy and verify the launch site. `viewModelScope` itself cancels correctly, so survival usually means the work escaped into another scope or entered code that never suspends and never checks cancellation. Durable work belongs behind a repository or `WorkManager`, with its lifetime stated explicitly.

WEAK ANSWER

“I would call `cancel()` from `onCleared()` because `viewModelScope` sometimes keeps running.”

INTERVIEWER PUSH

The job is no longer a child of `viewModelScope`. Which line detached it, and who cancels it now?

RECOVERY LINE

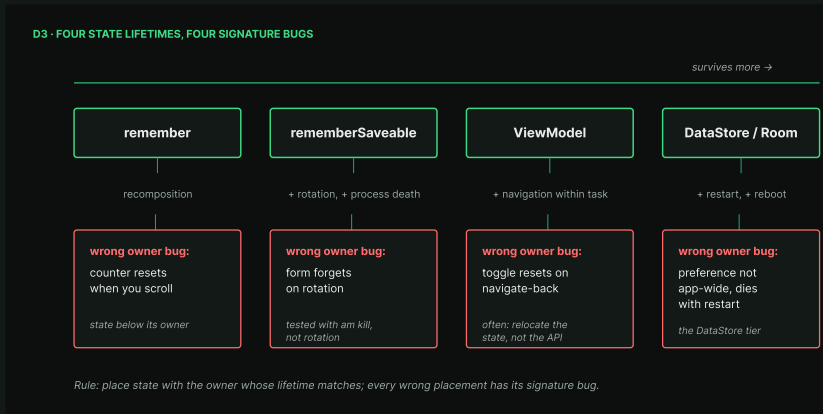
“I would inspect the job parent, remove the detached launch, and move intentionally durable work to an owner whose lifetime matches the requirement.”

WHAT EARNS THE POINT

The candidate identifies job ownership, cooperative cancellation, and the distinction between screen work and durable application work.

Q: Explain the difference between `remember` and `rememberSaveable`. In a real app, when would using the wrong one cause user-visible bugs after process death?

Must-know · Very common · Mid to senior



Choose a state owner by the lifetime the value must survive.

SHORT ANSWER

I use `remember` for state that only needs to survive recomposition. I use `rememberSaveable` for small UI values that must survive recreation, and durable owners for data that must be reconstructed after process death.

MEMORIZE THIS

The `remember` API survives recomposition; saveable state survives recreation within Bundle limits.

SENIOR ANSWER

The `remember` API survives recomposition only; it dies with the composition and with the process. RememberSaveable additionally writes into the saved instance state Bundle, so it survives configuration change and process death. Bugs appear when form input, scroll positions, or selected tabs use `remember`: the user backgrounds the app, the system kills the process, and their input vanishes on return. Custom types need a Saver. Anything large or non-serializable belongs in a ViewModel plus `SavedStateHandle` instead.

WHY THEY ASK

This question checks whether the candidate maps UI state to recomposition, configuration change, process recreation, and durable storage as separate lifetimes.

THE TRAP

Testing only rotation, which `remember` also fails, then assuming `rememberSaveable` is only about rotation. Stuffing large objects into `rememberSaveable` and hitting TransactionTooLargeException.

CODE

```
@Composable
fun NicknameField() {
    // Restored after recreation.
    var nickname by rememberSaveable { mutableStateOf("") }
    // Transient composition state.
    var isFocusedOnce by remember { mutableStateOf(false) }
    TextField(
        value = nickname,
        onValueChange = { nickname = it },
        modifier = Modifier.onFocusChanged { state ->
            if (state.isFocused) isFocusedOnce = true
        }
    )
}
```

FOLLOW-UP QUESTIONS

- Which values should never be placed in a `rememberSaveable` bundle?
- When should `SavedStateHandle` or a repository own the value instead?

LIVE INTERVIEW WORDING

I choose storage from the lifetime. A temporary expansion flag can stay remembered, form input may be saveable, and durable domain data should be reloaded from a ViewModel or repository. Saving everything into a Bundle is not resilience.

WEAK ANSWER

“I use `rememberSaveable` for every state value because it is a safer version of `remember`.”

INTERVIEWER PUSH

The value is a large domain object whose source of truth lives in a database. Why serialize it into the UI bundle?

RECOVERY LINE

“I would save only the minimal reconstructable UI input and reload durable domain data from its real owner.”

WHAT EARNS THE POINT

The candidate separates recomposition, recreation, process death, Bundle limits, and durable state ownership.

Buy the full book

The complete *24-Hour Android Interview Answer Book* contains:

- 160 senior Android answer cards across the full interview loop;
- the 50 questions most likely to decide the loop;
- 15 questions that separate senior judgment from memorized answers;
- 10 dangerous answers and their precise corrections;
- four complete technical, Compose, system-design, and incident simulations;
- the 24-hour plan, 2-hour emergency path, and morning-of checklist;

Get the complete edition at **mikesalari.dev/l/24hAndroid**.

Questions: **mike@salari.dev**