

MASTER KOTLIN FOR ANDROID INTERVIEW

— Learn Exactly What Android Interviewers Expect —

```
fun <T> flow(data: List<T>) =  
    flow {  
        for (item in data) {  
            emit(item)  
        }  
    }.flowOn(Dispatchers.IO)  
  
suspend fun fetchData(): Flow<Data> =  
    flow {  
        val result = api.getData()  
        emit(result)  
    }  
  
class MainViewModel : ViewModel() {  
    private val _state =  
        MutableStateFlow(State())  
    val state: StateFlow<State> =  
        _state.asStateFlow()  
}
```

INSIDE THIS BOOK

-  Deep Explanations
-  JVM & Compiler Internals
-  Android Production Use Cases
-  Senior Engineer Insights



Based on interview patterns from leading Product and Service-based technology companies.



146
Interview
Questions



100+
Diagrams



Interview-
Ready
Answers



Freshers →
Senior
Engineers

AZIM SIDDIQUI

KOTLIN • JVM • COROUTINES • FLOW • OOPS • COLLECTIONS

What's Inside

01	Kotlin Language Fundamentals Start here. Every interview begins with these. · 24 questions	p. 9
02	OOP In Kotlin Foundation of every Android architecture question. · 22 questions	p. 60
03	Null Safety Most common source of Android crashes. Interviewers love this. · 12 questions	p. 104
04	Scope Functions Asked in almost every Kotlin interview round. · 5 questions	p. 129
05	Collections & Functional Programming SDE2 must know all of these. · 17 questions	p. 140
06	Generics & Advanced Type System Senior-level differentiator. · 9 questions	p. 173
07	Coroutines — Core Concept Most asked section at SDE2 and above. Cannot skip. · 20 questions	p. 192
08	Coroutines — Advanced FAANG-level questions. Separates mid from senior. · 14 questions	p. 238
09	Flow & Reactive Streams Hot topic in every 2024-25 Android interview. · 15 questions	p. 268

10

Kotlin–Java Interop & JVM Annotations

p. 301

Relevant for companies with mixed Kotlin–Java codebases. · 8 questions

★

Quick-Reference Cheat Sheets

p. 318

Five one-page references — coroutines, Flow, collections, generics & interop

Every Question In This Book

All 146 questions, listed chapter by chapter with the page each one starts on — use this to jump straight to the topic you need.

01 Kotlin Language Fundamentals

Q1	What are the advantages of Kotlin over Java?	10
Q2	What is the difference between <code>var</code> , <code>val</code> , and <code>const val</code> ?	15
Q3	What is the advantage of using <code>const val</code> in Kotlin?	18
Q4	What is the difference between <code>Any</code> , <code>Unit</code> , and <code>Nothing</code> in Kotlin?	19
Q5	What is <code>Nothing</code> type and when does a function return <code>Nothing</code> ?	21
Q6	What is <code>typealias</code> and when would you use it?	23
Q7	What is the difference between <code>==</code> and <code>===</code> in Kotlin?	27
Q8	What are visibility modifiers in Kotlin?	29
Q9	What is the use of <code>internal</code> visibility modifier?	31
Q10	What is <code>StringBuffer</code> vs <code>StringBuilder</code> ?	33
Q11	What are named arguments and default parameters?	34
Q12	What are Lambda expressions in Kotlin?	36
Q13	What are Higher Order Functions in Kotlin?	38
Q14	What is an inline function in Kotlin?	39
Q15	What are <code>noinline</code> and <code>crossinline</code> keywords?	41
Q16	What is the <code>reified</code> keyword in Kotlin?	43
Q17	What is infix notation in Kotlin?	45
Q18	What are operator functions? Give real examples: <code>plus</code> , <code>invoke</code> , <code>get</code>	46
Q19	What is the <code>invoke</code> operator?	48
Q20	What are extension functions in Kotlin?	50
Q21	What are Labels in Kotlin?	52
Q22	What is destructuring declaration in Kotlin?	53
Q23		

	What is <code>componentN()</code> and how does it relate to destructuring?	55
--	--	----

Q24	What is the <code>by</code> keyword in Kotlin? Explain property delegation.	56
---------------------	---	----

02 OOP In Kotlin

Q25	What is a <code>data class</code> in Kotlin?	61
Q26	How is <code>data class</code> different from a regular class?	63
Q27	Common interview questions on <code>data class</code>	65
Q28	Can a <code>data class</code> extend another class?	67
Q29	What is a <code>data object</code> ? How is it different from <code>data class</code> ?	68
Q30	What is an <code>Init</code> block in Kotlin?	70
Q31	What are primary and secondary constructors in Kotlin?	72
Q32	What is the difference between <code>in</code> block and constructor?	74
Q33	How to make a singleton class in Kotlin?	76
Q34	Can we make a singleton class without the <code>object</code> keyword?	78
Q35	What is <code>companion object</code> in Kotlin?	80
Q36	What is the difference between <code>object</code> expression and <code>object</code> declaration?	82
Q37	What is the <code>open</code> keyword and why are Kotlin classes <code>final</code> by default?	84
Q38	What is the difference between <code>abstract class</code> and <code>interface</code> in Kotlin?	86
Q39	Can an <code>interface</code> have state in Kotlin?	88
Q40	What is the difference between <code>override</code> and <code>overload</code> ?	90
Q41	How does Kotlin handle multiple inheritance conflicts?	92
Q42	What is the difference between <code>is</code> , <code>as</code> , and <code>as?</code> in Kotlin?	94

01

CHAPTER 01 • THECODINGSHUF

Kotlin Language Fundamentals

| *Start here. Every interview begins with these.*

24

questions

Interview

ready answers

Android

production context

Q1

What are the advantages of Kotlin over Java?

Kotlin is preferred over Java in modern Android development because it helps you write code that is **safer, cleaner, more concise, and easier to maintain** — while staying fully compatible with existing Java code.

NOTE

- I would not say Java is bad. Java is mature, stable, and still used in many large Android codebases. But Kotlin solves many of the pain points Android developers face in day-to-day development.

STRONG INTERVIEW ANSWER

- Kotlin is preferred over Java for Android because it reduces boilerplate, improves null safety, supports modern asynchronous programming with coroutines, and works very well with Jetpack libraries and Compose. It also has full Java interoperability, so teams can migrate gradually instead of rewriting an entire project. From a practical Android development perspective, Kotlin improves productivity and reduces common runtime errors, which makes the codebase easier to maintain.

1. NULL SAFETY

One of the most common crashes in Android apps is `NullPointerException`. In Java, null handling depends mostly on developer discipline — nothing in the type system stops you:

```
JAVA

String name = null;
System.out.println(name.length()); // Crash at runtime
```

In Kotlin, nullability is part of the type itself:

```
KOTLIN

var name: String? = null
println(name?.length)           // prints null, cannot crash
```

Kotlin forces us to think about whether a value can be null. That catches many issues at compile time instead of letting them crash in production.

2. LESS BOILERPLATE

Java often requires a lot of code for simple models: constructors, getters, setters, `equals()`, `hashCode()` and `toString()`.

```

JAVA

public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

```

KOTLIN

data class User(val name: String)

```

With one line, Kotlin gives us a clean model class and automatically generates useful methods like `equals()`, `hashCode()`, `toString()` and `copy()`. That makes code easier to read and reduces mistakes.

3. COROUTINES FOR ASYNC WORK

Android apps constantly deal with background work: API calls, database operations, file handling and UI updates. In Java this usually needs callbacks, threads, executors or RxJava. Kotlin coroutines make async code easier to write and understand:

```

KOTLIN

viewModelScope.launch {
    val users = repository.getUsers()
    _uiState.value = UiState.Success(users)
}

```

Coroutines also support structured concurrency, cancellation and lifecycle-aware work, which is very useful in Android.

4. EXTENSION FUNCTIONS

Kotlin lets us add behaviour to existing classes without creating utility classes:



```

KOTLIN

fun String.isEmail(): Boolean {
    return contains("@")
}

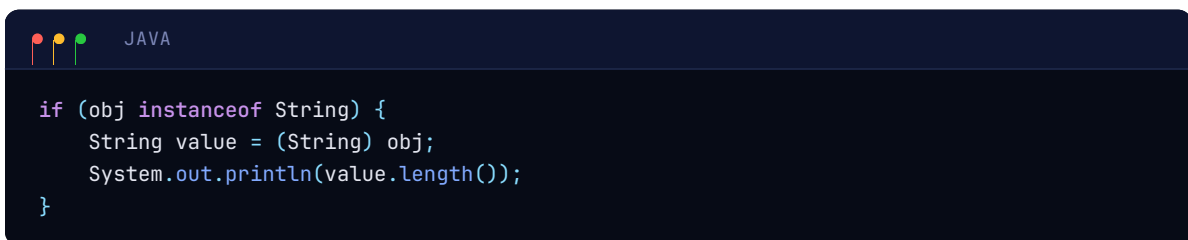
val valid = email.isEmail()

```

This keeps code readable and avoids large helper or utility classes.

5. SMART CASTS

Kotlin reduces unnecessary casting.



```

JAVA

if (obj instanceof String) {
    String value = (String) obj;
    System.out.println(value.length());
}

```



```

KOTLIN

if (obj is String) {
    println(obj.length)
}

```

Once Kotlin has checked the type, it automatically treats the object as that type.

6. BETTER SUPPORT FOR MODERN ANDROID

Most modern Android development is Kotlin-first:

- Jetpack Compose
- Coroutines and Flow
- Room, ViewModel and the rest of Jetpack
- Hilt
- Modern Android architecture samples

You can still use Java, but Kotlin feels far more natural with the current Android ecosystem.

7. JAVA INTEROPERABILITY

A very important point is that Kotlin works *with* Java, not instead of it. That means we can:

- Use Java libraries in Kotlin
- Call Kotlin code from Java
- Add Kotlin to an existing Java project
- Migrate feature by feature

This matters in real companies, because most teams cannot rewrite a full production app at once.

KOTLIN VS JAVA – FULL COMPARISON

AREA	JAVA	KOTLIN	WHY KOTLIN IS BETTER
Official Android support	Supported	Officially preferred for modern Android	Google recommends Kotlin for Android development
Boilerplate	More code required	Much less code	Faster development and easier maintenance
Null safety	Null errors usually found at runtime	Nullability checked at compile time	Reduces <code>NullPointerException</code> crashes
Data classes	Manual model classes, getters, setters, <code>equals()</code> , <code>hashCode()</code>	<code>data class User(val name: String)</code>	Cleaner model and data-layer code
Async programming	Threads, callbacks, executors, RxJava	Coroutines and Flow	Cleaner background work and lifecycle-aware code
Extension functions	Usually utility / helper classes	Add functions to existing classes	More readable and expressive code
Type casting	Manual casting required	Smart casts	Less repetitive and safer code
Default parameters	Requires method overloading	Built into the language	Fewer overloaded methods
Functional programming	Available, but verbose on Android	Lambdas and collection operators are concise	Cleaner list transformations and event handling

AREA	JAVA	KOTLIN	WHY KOTLIN IS BETTER
UI development	XML / View system works well	Best fit for Jetpack Compose	Compose is Kotlin-first
Sealed classes	More verbose alternatives	Built-in sealed classes and interfaces	Great for UI states and result handling
Interoperability	Can work with Kotlin	Fully interoperable with Java	Easy gradual migration
Learning curve	Easier if you already know Java	Slight learning curve	Worth it for modern Android productivity

Final Mental Model

From a senior Android perspective, I would choose Kotlin for new development because it improves **safety, productivity and maintainability**. But I would not blindly rewrite stable Java code.

In a real project, I would introduce Kotlin gradually, use it for new features, and migrate older Java code only when there is a clear benefit — or when that area is already being changed.

Interview-Ready Answer

Kotlin is preferred over Java in Android because it makes code safer, shorter and easier to maintain. Its null safety helps prevent common crashes, data classes reduce boilerplate, coroutines simplify asynchronous programming, and features like extension functions, smart casts, default parameters and sealed classes make everyday Android development cleaner. It is also fully interoperable with Java, so teams can adopt it gradually. That combination of safety, productivity and compatibility is why Kotlin is the recommended language for modern Android development.

Q2

What is the difference between `var`, `val`, and `const val`?

In Kotlin, variables are declared mainly using `var`, `val`, and `const val`. The difference between them depends on whether the value can change and when the value is known.

VAR – MUTABLE VARIABLE

`var` is used for a mutable variable. This means the value can be changed after it is assigned. For example:

```
KOTLIN
var count = 0
count = count + 1
```

Here, `count` can change, so `var` is correct. Use `var` when the value is expected to change, such as loading state, counter, score, selected item, etc.

VAL – READ-ONLY VARIABLE

`val` is used for a read-only variable. Once a value is assigned, it cannot be reassigned.

```
KOTLIN
val name = "Azim"
name = "Rahul" // Error
```

Use `val` when the value should not change after assignment. In Kotlin, it is a good practice to use `val` by default and use `var` only when modification is required.

One important point is that `val` stops reassignment, but it does not always stop internal object changes.

```
KOTLIN
val list = mutableListOf("A", "B")
list.add("C") // Allowed
```

Here, `list` still points to the same object, so adding data is allowed. But assigning a new list is not allowed.

```
KOTLIN
list = mutableListOf("X") // Error
```

CONST VAL – COMPILE-TIME CONSTANT

`const val` is used for compile-time constants. This means the value must be known before the app runs.

```
KOTLIN

const val BASE_URL = "https://api.example.com/"
const val MAX_RETRY = 3
```

`const val` is generally used for fixed values like base URLs, API keys, preference keys, database names, intent keys, and log tags. It can only be used with basic types such as `String`, `Int`, `Boolean`, `Long`, `Double`, `Float`, and `Char`.

Unlike `val`, `const val` cannot take values from function calls because function results are known only at runtime.

```
KOTLIN

val time = System.currentTimeMillis() // Allowed
const val time = System.currentTimeMillis() // Error
```

Also, `const val` cannot be declared inside a normal function. It should be declared at the top level, inside an `object`, or inside a `companion object`.

```
KOTLIN

object AppConstants {
    const val BASE_URL = "https://api.example.com/"
}
```

COMPARISON TABLE

KEYWORD	MEANING	CAN CHANGE?	VALUE KNOWN AT	EXAMPLE USE
<code>var</code>	Mutable variable	Yes	Runtime	Counter, loading state
<code>val</code>	Read-only variable	No reassignment	Runtime	User object, fixed value
<code>const val</code>	Compile-time constant	No	Compile time	Base URL, keys, tags

Interview-Ready Answer

In Kotlin, `var` is mutable, so its value can be changed. `val` is read-only, so once assigned, it cannot be reassigned. `const val` is also read-only, but it is a compile-time constant, so its value must be known before the app runs. `var` and `val` can be declared inside functions, but `const val` must be declared at top level, inside an object, or inside a companion object.

Q3

What is the advantage of using `const val` in Kotlin?

WHAT IT IS

`const val` declares a value the **compiler** can see, not just the runtime. It must be a primitive or `String`, and must sit at the top level or inside an `object` / `companion object`.

WHY IT MATTERS

Annotations are processed at compile time, so they can only accept compile-time constants. This is not a style preference — a plain `val` simply **will not compile** here:

```
KOTLIN

@Entity(tableName = UserEntity.TABLE_NAME)
data class UserEntity(
    @PrimaryKey
    @ColumnInfo(name = UserEntity.COLUMN_ID)
    val id: String
) {
    companion object {
        const val TABLE_NAME = "users" // const val required here
        const val COLUMN_ID = "user_id"
        // val TABLE_NAME = "users" // compile error in @Entity
    }
}
```

The same applies to WorkManager unique-work names, Intent extra keys, and Retrofit path segments — anywhere a value has to be visible before the app ever runs.

COMMON CONFUSION

▲ PUBLIC `CONST VAL` IS PART OF YOUR BINARY API

- The compiler **copies the value** into every file that reads it — it is not a field lookup.
- Ship `const val API_VERSION = 1`, later change it to `2`, and modules compiled against the old version still see `1` until they are recompiled.
- Rule of thumb: `const val` for values that are genuinely permanent. Use a regular `val` or a function for anything that might evolve.

Interview-Ready Answer

`const val` makes the value available at compile time, so the compiler can inline it. That is what lets it be used inside annotations — Room's `@Entity(tableName = ...)` and `@ColumnInfo(name = ...)` require compile-time constants, and a plain `val` fails there because it is initialized at runtime. The trade-off worth mentioning is inlining: a public `const val` in a library becomes part of your binary API, because consumers bake the value into their own bytecode.

Q4 What is the difference between Any, Unit, and Nothing in Kotlin?

WHAT IT IS

Three types that together make Kotlin's type system complete — they cover every possible outcome a computation can have.

TYPE	MEANING	HAS INSTANCES?
<code>Any</code>	Root of all non-null types — Kotlin's <code>Object</code>	Yes
<code>Unit</code>	The function completed, but produced nothing useful	Yes — exactly one (<code>Unit</code>)
<code>Nothing</code>	The function never completes normally — it throws or loops forever	No — none, ever

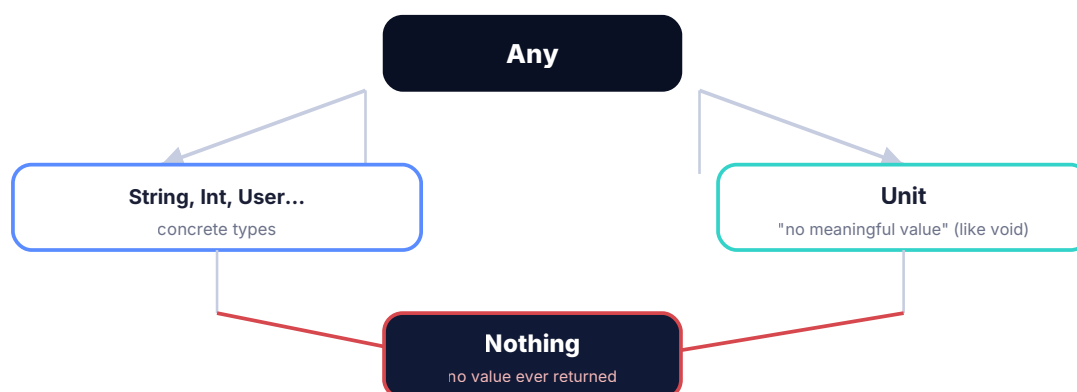


Figure · Any / Unit / Nothing in the Kotlin type hierarchy

HOW IT WORKS INTERNALLY

`Any` maps straight to `java.lang.Object`. `Unit` is a real singleton object, though the compiler usually optimizes it away at call sites so it behaves like `void`.

`Nothing` has **no JVM representation at all**. It is a pure compiler concept used to model unreachable code. Because it has no instances, it is safe to treat it as a subtype of *every* type — which is exactly why `throw` can appear anywhere an expression is expected.

COMMON CONFUSION

▲ ANY IS NOT THE TOP OF THE HIERARCHY — ANY? IS

- `Any` excludes `null`. The true root type, covering nullable types too, is `Any?`.
- `Unit` and `Nothing` are not interchangeable: `Unit` means *finished with no useful value*; `Nothing` means *never finished at all*.
- `TODO()` returns `Nothing` — it compiles anywhere, then throws `NotImplementedError` in production.

Final Mental Model

`Any` = *anything can be here*. `Unit` = *something happened, nothing came back*. `Nothing` = *control never gets here at all*.

Interview-Ready Answer

`Any` is the root of all non-null types — the equivalent of Java's `Object`, with `Any?` sitting above it to include null. `Unit` is the type of a function that completes without a meaningful result; it is a real singleton, not just `void`, which is what lets lambdas and generics work uniformly. `Nothing` is the type of a computation that never returns — it always throws or loops. It has no instances, so it is a subtype of every type, and that is what allows `throw` to sit inside an Elvis expression or a `when` branch and still type-check.

Kotlin Interview Cheat Sheet

VARIABLES

- `var` — reassignable
- `val` — read-only reference (deep mutability still possible)
- `const val` — compile-time constant, top-level/object only

COLLECTIONS

- `map` transform, `filter` select, `reduce/fold` accumulate
- `flatMap` flattens nested collections after mapping
- `Sequence` — lazy, better for large/chained pipelines

NULL SAFETY

- `?.` safe call — null-safe chaining
- `?:` Elvis — fallback value
- `!!` not-null assertion — throws NPE, avoid in production
- `lateinit` for var, `by lazy` for val (thread-safe by default)

COROUTINES

- `launch` fire-and-forget, `async` returns a `Deferred`
- Dispatchers: `Main` (UI), `Default` (CPU), `IO` (network/disk)
- `viewModelScope` / `lifecycleScope` over `GlobalScope`
- `SupervisorJob` — one child failing doesn't cancel siblings

SCOPE FUNCTIONS

- `let` — transform / null-check, uses `it`
- `run` / `with` — compute a result, uses `this`
- `apply` — configure and return the object, `this`
- `also` — side effects, returns the object, `it`

FLOW

- `Flow` is cold; `StateFlow` / `SharedFlow` are hot
- `StateFlow` always holds one current value
- `debounce`, `buffer`, `conflate` shape back pressure

OOP

- `data class` — equals/hashCode/copy generated
- `sealed class` — restricted hierarchy, exhaustive `when`
- `enum class` — fixed singleton instances
- classes are `final` by default — use `open` to allow override

KOTLIN ↔ JAVA INTEROP

- `@JvmStatic` — expose companion member as real static
- `@JvmOverloads` — generate overloads for default params
- `@Throws` — declare checked exceptions for Java callers

Generics, Variance & JVM Interop Cheat Sheet

VARIANCE

- `out T` — covariant, producer-only
- `in T` — contravariant, consumer-only; subtyping reverses
- PECS: Producer `extends (out)`, Consumer `super (in)`

@JVMSTATIC / @JVMFIELD

- `@JvmStatic` — real static for a companion member
- Top-level functions are **already** static — live in `FileKt`
- `@JvmField` — raw field, no getter, no encapsulation

STAR PROJECTION

- `List<*>` — some specific unknown type, not "any type"
- Reading gives `Any?`; writing is always forbidden
- Not Java's raw type — type safety is preserved

@JVMOVERLOADS / @JVMNAME

- `@JvmOverloads` — generates overloads right-to-left
- Essential on custom `View` constructors for XML inflation
- `@JvmName` — renames file class, resolves erasure clashes

TYPE ERASURE & REIFIED

- Generic args vanish at runtime — `List<String> == List<User>`
- `reified` only works on `inline` functions
- A reified `List<User>` still erases the `User`

CHECKED EXCEPTIONS

- Kotlin has none — every exception is unchecked
- `@Throws` — restores `throws` for Java callers only
- Prefer a sealed `Result` type over relying on exceptions

VALUE CLASSES

- `@JvmInline value class` — distinct type, erased at runtime
- Boxes in generics, as nullable, or upcast to an interface/ `Any`
- Win is correctness, not raw performance

EXTENSION FUNCTIONS

- Compile to static methods; receiver becomes parameter zero
- Dispatched on the declared type — not polymorphic
- Cannot access private members; a real member always wins

THECODINGSHF

Thank You

You made it through 146 questions written the way I wish someone had written them for me — not as definitions to memorise, but as the trade-offs, internals, and production scars that actually come up when a Staff engineer is on the other side of the table.

If even a handful of these pages changed how you think about a `suspend` function, a `StateFlow`, or what `data class` is really generating for you, this book did its job. That was always the bar — not page count, not coverage, but whether it was worth your time.

If it helped, the most useful thing you can do is tell one other engineer preparing for the same interviews. And if something in here was unclear, wrong, or could be sharper, I'd genuinely like to hear about it — email or LinkedIn, both below.

Good luck in the room. Go be the person who knows why, not just what.

Azim Siddiqui

Android Engineer · TheCodingShf

hello@thecodingshef.in thecodingshef.in linkedin.com/in/azimsidd