Kotlin Mastery for Java Developers

Below is a **6-day (9:00–16:00) Kotlin course syllabus** tailored for experienced Java developers. The focus is not on "Kotlin syntax over Java syntax", but rather on how Kotlin enables more **idiomatic, safe, expressive, and coroutine-friendly** programming.

Daily Schedule :

09:00–09:15	Kickoff / Review
09:15–10:45	Lecture + Code Examples
10:45–11:00	Break
11:00–12:30	Hands-on Practice
12:30–13:30	Lunch
13:30–15:00	Advanced Concepts + Labs
15:00–16:00	Code Transformation + Group Exercises

Day 1: Kotlin Foundations for Java Minds

Goal: Unlearn Java habits. Rebuild thinking with Kotlin's syntax and control flow.

Topics:

- **Kotlin Philosophy vs Java:** conciseness (strip boilerplate via type inference, data classes, default/named args); safety (null-safety operators, smart casts, requireNotNull); idiomatic expressiveness (scope functions, functional collections).
- **Kotlin Basics:** type inference; val vs var immutability; null-safety (?., !!, Elvis ?:, requireNotNull); compiler flow-analysis for smart casts.
- **Control Flow:** if/when as expressions; ranges and progressions (.., downTo, until, step); smart casts after is or null checks.
- **Collections & Scope Functions:** read-only vs mutable List/Map/Set; array factories (arrayOf, intArrayOf); scope functions (let, run, apply, also, with) for concise object setup.
- **Kotlin Functions:** default arguments & named parameters; expression-bodied functions; infix functions; extension functions for fluent DSLs.
- **Basic OOP:** concise classes with primary-constructor properties; init blocks; visibility modifiers (public, internal, private); open/abstract inheritance; interfaces with default implementations; safe (as?) vs unsafe (as) casting.
- **Data Classes & Destructuring:** data class auto-generates equals/hashCode/toString/copy/componentN; destructuring declarations for unpacking.

- **Error Handling:** try/catch/finally as expressions; no checked exceptions; Java interop via @Throws; use { ... }for automatic resource closing.

Exercises:

- Null-Safety Refactoring: eliminate nested null checks by using safe-call (?.), Elvis (?:), ?.let { }, and leverage smart casts to handle nullable types in one expression.
- Overloadst o Default Args & Java Interop: collapse multiple log overloads into one function with default and named parameters; use @JvmOverloads (or @JvmName) so Java callers still work.
- POJO+Builder into Data Class & DSL: replace manual Builder pattern with a data class; add extension properties, destructuring (componentN()), a when-based age classifier, and an infix years DSL helper.
- Scoping Functions for Init: streamline multi-step object creation using apply for configuration, run for computed returns, and also for side-effects (e.g., logging) in a single expression.
- Early Validation & Fail-Fast: validate inputs up-front with requireNotNull and Elvis-return (?: return); throw IllegalArgumentException on failure; default values via ?: and convert to an expression-bodied function.
- when + Smart Casts: refactor nested if/else chains into a single when(event) expression; rely on compiler smart casts inside each branch and include an else for unknown cases.
- Nullable List Cleanup: process chaining filterNotNull(), mapNotNull, and summing results in a concise functional pipeline.

Day 2: Idiomatic Kotlin: Functions, Lambdas & Collections

Goal: Develop functional fluency and Kotlin-style APIs.

Topics:

- **Lambda-Expression Syntax & Memory Model**: Full syntax variants, implicit it, trailing-lambda, last-expression return, compiler generates invoke() classes, capture costs, non-capturing vs capturing lambdas.
- **High-Order Functions**: Definition, standard-library examples (run, apply, map, filter), function references (::foo, MyObj::bar) – syntax and use-cases, Local vs non-local returns; inline, noinline, crossinline.
- **Inline Functions & Reified Generics**: Allocation-free call-sites; limitations on private/internal access, Reified T for reflection-free JSON-decode, DI look-ups, Intent extras.
- **Value (Inline) Classes & Type Aliases**: Zero-overhead domain types (UserId, Meters), validation vs. pure aliases.
- **Extension Properties**: Adding behaviour or computed data to existing class cobining with Refied parametrs

- **Generics Deep-Dive**: **Variance**: out (covariance), in (contravariance) with Cat/Animal examples, Use-site wild-cards versus declaration-site variance, **Star-projection** List<*> for “unknown but safe” read-only views, Full Java comparission
- **Functional Collections**: Core transform/test/aggregate ops (map, flatMap, mapIndexed, filterIsInstance, fold, sumOf, ...), Set-style ops (distinct, union, intersect), take/drop, zip, takeIf.
- **Lazy Sequences**: asSequence(), generateSequence, sequence { yield ... } (prime-number and Fibonacci demos), Performance proof: eager vs lazy pipeline timing, advance algorithmic drill using sequnce

Exercises:

- **Lazy Sequence Summation:** write an inline mapSumOf to sum a filtered sequence without intermediate lists; compare its speed/allocations to the original stream-based version.
- **Lambda Performance:** invoke a million-iteration function with four lambda styles (non-capturing, capturing, reference, anonymous) and time each to observe capture overhead.
- **Retry HOFs:** build an inline retry that accepts a noinline error handler; add a crossinline variant to illustrate return restrictions.
- **Reified Decoder:** create an inline, reified JSON decoder that instantiates any class via its no-arg constructor.
- **Units as Value Classes:** replace a Double alias with a @JvmInline value class; add a converter extension and enforce unit safety in function signatures.
- **Reified tag Property:** define a reified extension property that yields Type@hash strings for any object.
- **Variance with Box:** implement feed(box: Box<out Animal>), stock(box: Box<in Tiger>, cub), and inspect(box: Box<*>) to demonstrate out, in, and star-projection.
- **Grades Digest Drill:** in one expression, group marks by course to compute averages and identify the top-averaging student.
- **E-commerce Analytics Pipeline:** craft immutable pipelines using core collection operators to produce customer spend maps, “whales,” Pareto SKU sets, country dashboards, and customer breadth metrics.

Day 3: Object Orientation Done the Kotlin Way

Goal: Break away from Java-style class hierarchies.
Topics:

- **Deferred & Delegated Initialization** - lazy {}, lateinit, init {}, Property delegates: by lazy, Delegates.notNull, custom.
- **Singleton & “Static” Patterns** - object declaration vs. object expression vs. companion object, @JvmStatic bridges, Interface / property delegation with by.

- **Callable Objects & Functional Interop** - SAM conversion, fun interface, Implementing function types class C : (A) → B, operator fun invoke for factory/wrapper objects.
- **Operator Overloading** - Arithmetic and collection operators (plus, get, set), Compound forms plusAssign, minusAssign; immutable-collection contract.
- **Nested vs Inner & Delegation vs Inheritance -** Scoping, memory-leak rules, when to delegate instead.
- **Algebraic-Data Types** - Sealed class / interface for closed variants, Value / data classes as product types, Exhaustive when.
- **DSL Construction Toolkit -** Lambdas with receiver, extension + infix functions, +=, End-to-end mini-DSL demo tying all features together.
- **Context Receivers (1.9+) -** Multi-receiver blocks with context(A, B), Android examples: Nav + Lifecycle, Compose Theme + Density.

Exercises:

- **Encrypted SharedPreferences (Class Delegation):** wrap a SharedPreferences instance via by prefs; override getString/ getStringSet to decrypt on read (preserving nulls) and return new immutable collections; override edit() to return an Editor wrapper that encrypts in putString/putStringSet before delegating—preserving method chaining.
- **Tiny DSL Form Validation:** build a field<T>(name) { ... } DSL using lambda-with-receiver to compose Rule<T>s (notBlank, contains, atLeast) with infix and/or; assemble fields in form { +email; +age }; implement operator fun invoke(data: Map<String,Any?>) to return success or a map of human-readable errors; support type-checking, missing-key errors, and optional String.invoke/Int.invoke sugar with distinct @JvmName.
- **Simple Text-Speaking App:** enable ViewBinding with lazy ActivityMainBinding.inflate(layoutInflater) to setContentView(root); wire Button click first via object expression then SAM lambda and SeekBar via object expression; implement MyTextToSpeech wrapper (implements OnInitListener & UtteranceProgressListener, ctor with default callbacks, play, stopTalking, operator fun invoke); integrate by creating a lazy speech instance and invoking it on Button press with EditText text; add lifecycle cleanup, error handling, and small extension helpers.

Day 4: Coroutines and Asynchronous Thinking

Goal: Learn structured concurrency and callback refactoring.
Topics:

- **Structured Concurrency Essentials:** CoroutineScope (including lifecycleScope, viewModelScope, ServiceLifecycleDispatcher), launch/

async (with start = LAZY), withContext, runBlocking; Job.join/joinAll; cooperative cancellation.

- **Coroutines vs Threads:** suspension vs blocking; dispatchers (Default, IO, Main, Unconfined) and custom contexts (newSingleThreadContext) mapping to thread pools.
- **Callback-to-Suspend Refactoring:** wrap listener or completion-handler APIs with suspendCoroutine/suspendCancellableCoroutine and invokeOnCancellation cleanup.
- **Synchronous Lazy Streams:** Sequence builder (yield/yieldAll) for pull-based, synchronous pipelines.
- **Cold & Reactive Streams:** define cold Flows; use flowOn; apply map/filter/transform/debounce; handle errors with retryWhen/catch; inject side-effects via onStart/onEach/onCompletion; flatten with flatMapLatest.
- **Hot Streams & Buffering:** StateFlow vs SharedFlow; share with stateIn/shareIn and SharingStarted policies; tune back-pressure via buffer/conflate/collectLatest; use Channels for FIFO queues.
- **Event Streams:** adapt multi-event callbacks or loops into cold, cancel-aware Flows with callbackFlow and awaitClose cleanup.
- **Flow vs Suspend Guidelines:** choose single-shot suspend functions for one-off results and Flows for streams requiring back-pressure, cancellation, or composition.
- **Lifecycle & Cancellation:** attach scopes to Android components or server jobs; suspend until lifecycle states (whenCreated/whenStarted/whenResumed); ensure cleanup in finally or awaitClose.

Exercises:

- Callback to Suspend: wrap a listener-based Java API with suspendCancellableCoroutine, trigger it, then cancel to verify cleanup.
- Thread Loop to Flow: turn an ExecutorService/while-loop producer into a cold callbackFlow and add debounce + retryWhen.
- Permission Wrapper: expose Android permission requests as a one-shot suspend function and a status Flow.
- Activity For Result Wrapper, Wrap Speech to text with suspendCoroutine
- StateFlow Registry: build a small in-memory registry that publishes live updates via MutableStateFlow and is collected on UI scope.
- Create a real app flow with Flow to UI updates with whole pipline while doing transformation and mnipulation on flow
- Wrap Android events callback such as network conncetivity and battry chjanged with flow and propogate changes.
- **ASR Integration & Permission Handling:** implement a PermissionRequest class using ActivityResultRegistry and suspendCancellableCoroutine to request RECORD_AUDIO; build a SpeechToText wrapper that uses suspendCoroutineto launch

RecognizerIntent and return the best-match transcript via operator fun invoke().

- **Reactive Flow UI Pipeline:** expose EditText changes as a cold Flow with callbackFlow; in the ViewModel apply debounce, filter, distinctUntilChanged, flatMapLatest, and retryWhen; convert to StateFlow via stateIn and collect lifecycle-aware in the UI to drive a RecyclerView from a repository Flow<List<Stock>>.

Day 5 – Production Foundations (Testing, DI, Error-Flow, Interop)

Goal: Give developers the must-have tools for production Kotlin on Android/JVM: rock-solid unit tests, clean DI and typed error channels.

Topics:

- **Testing Foundations (Kotest):** Running Kotlin tests on the JUnit Platform; choosing Kotest spec styles (FunSpec, StringSpec, BehaviorSpec, etc.); expressive DSL test naming; dynamic tests; mixing JUnit and Kotest in one project.
- **Coroutines & Flow Testing:** Deterministic tests with runTest and virtual time; StandardTestDispatcher vs UnconfinedTestDispatcher; advanceTimeBy, advanceUntilIdle, runCurrent; Flow assertions with Turbine (awaitItem, awaitComplete, expectNoEvents) and handling optional intermediate emissions.
- **Mocking in Kotlin:** MockK for suspend functions and final classes (coEvery, coVerify, object/top-level mocking) vs Mockito trade-offs; argument matchers, relaxed mocks, and verification patterns.
- **Property-Based Testing:** Using Kotest generators (Arb.*) and checkAll to validate invariants across many randomized inputs; shrinking and counter-examples; when to prefer properties over examples.
- **Dependency Injection with Hilt (Dagger):** Why DI; constructor vs field injection; @HiltAndroidApp, @AndroidEntryPoint, @HiltViewModel; modules with @Binds and @Provides; qualifiers; component scopes (Singleton, ViewModelScoped); replacing modules in tests (@HiltAndroidTest, @TestInstallIn, @UninstallModules, @BindValue).
- **Architecture & Compose Glimpse:** Clean layering (UI → Presentation → Domain → Data); ViewModel + StateFlow as UI state; Compose interop (collectAsState, remember, LaunchedEffect); mapping domain models to UI and triggering use-cases from the VM.

Exercises:

- Build production-ready confidence in Kotlin coroutines/Flow by testing end-to-end with Turbine (precise emissions & timing), MockK (isolated IO + verifications), Kotest property tests (input laws), and coroutines-test (virtual time), proving cancellation (flatMapLatest), cache vs. network paths, and robust StateFlow UI states—flake-free and CI-

friendly.
- Refactor a Room-backed MVVM app into a **decoupled, testable architecture** by replacing hard-wired dependencies with **Dependency Injection via Hilt** (bind interfaces, provide modules & scopes, inject DAOs/Repositories/ViewModels), improving modularity, lifecycle scoping, and testability—built collaboratively in class.
- Use-Case Pipeline Test: compose three callable use-cases with operator invoke, run them in parallel with coroutines, and unit-test with runTest.
- Migrate a full Java use case to idiomatic Kotlin - **Mini Capstone Migration** – Take a small Java feature (controller + service) and port it to Kotlin clean-architecture style: operator fun invoke use-case, coroutines, Either for errors, and JSON serialization with kotlinx.serialization.

Day 6 - Under the Hood (Native Bridges, JVM dive, Deep Optimisation, Seamless Java Interop)

Topics

- **JNI/NDK in Kotlin:** external funcs, System.loadLibrary, JNI call path & JNI_OnLoad, value/inline-class caveats, null-safety across JNI, wrap native callbacks with suspendCancellableCoroutine.
- **Async & State:** sealed-state models for progress/error, cancellation & timeouts, keep orchestration in Kotlin; thin JNI layers.
- **Kotlin Reflection vs Java:** kotlin.reflect from @Metadata, nullability/ defaults, isSuspend, sealedSubclasses, when to prefer Java reflection and javaMethod.
- **JVM Optimization Basics:** class loaders, MethodHandle vs reflection, how lambdas compile (invokedynamic), escape analysis & object allocation tips.
- **Java Interop:** @JvmStatic, @JvmOverloads, @JvmName/file-level, @JvmSuppressWildcards, @JvmSynthetic; clean mixed-language APIs.
- **Kotlin Multiplatform:** commonMain + expect/actual, shared use-cases/DTOs, threading notes; where native C/C++ still fits.

Exercises

- **Native-Zlib Spike:** Hands-on exercise to build a JNI bridge to Android's built-in zlib for compression/decompression, wrap it in coroutine-friendly Kotlin APIs, and verify via a simple UI—giving students practical experience with native integration, CMake configuration, and off-main-thread coroutine dispatch for CPU-bound work.
- **Kotlin Reflection Deep-Dive:** hands-on lab to build a "Reflection Playground" that inspects KClass metadata, discovers and mutates properties (even private), invokes functions (with defaults and private),

explores constructors, handles top-level & extension functions, dispatches via annotations, inspects generic types, and wires a mini-DI container—equipping students with practical mastery of runtime Kotlin reflection.

- JVM Optimisation Micro-Lab — write a tiny custom class-loader that reloads a plugin class; compare reflective call vs MethodHandle invocation.
- Expose a Kotlin API to Java by applying the right @JvmStatic, @JvmOverloads, @JvmField, @JvmName and Verify Java can call it with lambdas via the SAM adapter.