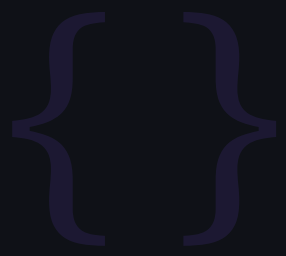

KOTLIN

Intermediate

Coroutines Basics & Advanced · Generics & Variance

Delegation · DSL Building

Reflection & Annotations · Advanced Patterns

Kotlin Multiplatform Basics

Philip Osztromok

Generated with Claude

Table of Contents

Kotlin Intermediate — 8 Chapters

🌀 Coroutines Basics

Coroutines are Kotlin's approach to asynchronous programming — lightweight, suspendable computations that let you write async code that reads like straight-line, sequential code, instead of nested callbacks. This chapter covers suspend functions, CoroutineScope, the launch and async builders, and Dispatchers — the vocabulary every later chapter in this course builds on.

📦 Setup: kotlinx.coroutines

Unlike `val`/`var` or null safety, coroutines aren't built into the core language syntax alone — `launch`, `async`, and `Dispatchers` come from the **kotlinx.coroutines** library (the `suspend` keyword itself *is* a language feature; the library provides the tools that actually run suspend functions). In a real project this is a Gradle dependency:

```
// build.gradle.kts
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.0")
}
```

It ships pre-added in Android Studio projects and most Kotlin server templates, so this is usually a one-time setup step rather than something to think about chapter to chapter.

⏸ suspend Functions

A function marked `suspend` can be paused and resumed without blocking the thread it's running on:

```
suspend fun fetchUserName(): String {
    delay(1000)  // suspends for 1 second — does NOT block the thread
return "Philip"
}
```

`delay()` is the coroutine world's non-blocking equivalent of `Thread.sleep()` — it pauses the coroutine, freeing the underlying thread to do other work in the meantime, then resumes exactly where it left off once the delay is up. A `suspend` function can only be called from inside **another suspend function**, or from inside a coroutine — the compiler enforces this, so there's no way to accidentally call a slow suspending operation from ordinary, non-suspending code.

⚠ suspend Doesn't Mean "Runs on a Background Thread"

Marking a function `suspend` only means it *can* pause without blocking — it says nothing about which thread it runs on. That's a separate decision, controlled by **Dispatchers** (covered later in this chapter). A common beginner assumption is that `suspend` alone moves work off the main thread; it doesn't.

Sequential, Not Callback-Nested

Two suspend calls in a row (`val a = fetchA(); val b = fetchB()`) read top-to-bottom, exactly like synchronous code — no `.then()` chains, no nested callbacks, even though both calls are genuinely asynchronous under the hood.

vs JavaScript async/await

This will feel immediately familiar coming from JavaScript's `async`/`await` — `suspend fun` plays the role of an `async function`, and calling a suspend function reads like `await`ing a promise, without a separate keyword at the call site.

🌐 CoroutineScope

A coroutine can't just be launched into the void — every coroutine belongs to a **CoroutineScope**, which defines its lifetime and ties it to a specific place in the code that "owns" it:

```
fun main() = runBlocking {
    // "this" here is a CoroutineScope, provided by runBlocking
    launch {
        delay(500)
        println("Coroutine finished")
    }
    println("main() continues immediately")
}
```

`runBlocking` is a bridge between ordinary blocking code (like `main()`, or a test function) and the coroutine world — it blocks the current thread until every coroutine launched inside it completes. It's mainly a tool for `main()` functions and tests; real applications almost never use `runBlocking` elsewhere, since blocking a thread is exactly what coroutines exist to avoid.

⚠ Structured Concurrency

Every coroutine launched inside a scope is tied to that scope's lifetime — if the scope is cancelled, every coroutine inside it is cancelled too, automatically. This is called **structured concurrency**, and it's the core safety guarantee coroutines provide: there's no such thing as an orphaned coroutine silently running forever with nothing tracking it, unlike a raw background thread started and forgotten about.

🚀 launch — Fire and Forget

`launch` starts a new coroutine that runs independently and doesn't return a usable result — it's for work you care about happening, not work you need a value back from:

```
fun main() = runBlocking {
    val job = launch {
        delay(1000)
        println("Task done")
    }
    println("Task launched")
    job.join()  // suspend here until the launched coroutine finishes
}
// Output:
// Task launched
// Task done      (printed ~1 second later)
```

`launch` returns a `Job` — a handle representing the running coroutine. `job.join()` suspends the current coroutine until that job completes (without blocking the thread), and `job.cancel()` stops it early. If nothing ever calls `join()`, `launch` is genuinely "fire and forget" — the calling code moves on immediately while the coroutine runs in the background.

🔁 async / await — Getting a Result Back

`async` is `launch`'s counterpart for when you *do* need a value back — it returns a `Deferred<T>`, which is Kotlin's name for what other languages call a Future or Promise:

```
suspend fun fetchScore(): Int {
    delay(500)
    return 42
}

fun main() = runBlocking {
    val deferred = async { fetchScore() }
    println("Waiting...")
    val result = deferred.await()  // suspends until the result is ready
    println("Score: $result")
}
```

`.await()` is the direct Kotlin equivalent of JavaScript's `await`. The real payoff of `async` is running independent tasks **concurrently**, then collecting both results:

```
fun main() = runBlocking {
    val deferred1 = async { fetchScore() }  // starts immediately, runs concurrently
val deferred2 = async { fetchScore() }  // starts immediately too, doesn't wait for deferred1
val total = deferred1.await() + deferred2.await()
    println("Total: $total")  // takes ~500ms total, not ~1000ms — both ran at the same time
}
```

launch vs async

	launch	async
Returns	Job	Deferred<T>
Result value?	No	Yes, via await()
Use case	"Do this" — no result needed	"Compute this" — need the value back
Unhandled exception	Crashes immediately	Held until await() is called

🪡 Dispatchers — Choosing Where Code Runs

A `Dispatcher` decides which thread (or thread pool) a coroutine actually runs on:

```
launch(Dispatchers.Default) {
    // CPU-intensive work — sorting, parsing, calculations
}

launch(Dispatchers.IO) {
    // Blocking I/O — network calls, file/database reads
}

launch(Dispatchers.Main) {
    // UI updates — Android's main thread, or similar UI-toolkit thread
}
```

Dispatchers.Default

A thread pool sized to the number of CPU cores, meant for CPU-bound work that genuinely keeps a processor busy — computation, not waiting.

Dispatchers.IO

A larger, elastic thread pool meant for blocking I/O calls that spend most of their time waiting, not computing — network requests, file access, database queries.

Dispatchers.Main

Only available with a UI framework on the classpath (e.g. Android). Confined to the single UI thread — required for touching UI elements, and where a coroutine typically starts and ends even if it dispatches work elsewhere in between.

withContext(...)

Switches the dispatcher for a block of code within an already-running coroutine — e.g. do a network call on `Dispatchers.IO`, then `withContext(Dispatchers.Main)` to update the UI with the result.

Coroutines vs Java Threads vs JavaScript's Event Loop

	Java Threads	JavaScript	Kotlin Coroutines
Unit of concurrency	OS thread (heavyweight, ~1MB stack)	Single thread + event loop	Lightweight, thousands run per thread
Blocking a "slot"?	Yes — a blocked thread is a wasted thread	N/A — no blocking, only one thread	No — suspend frees the thread while waiting
True parallelism?	Yes, across cores	No, single-threaded	Yes, via Dispatchers.Default/IO thread pools
Cancellation	Cooperative, awkward (interrupt flags)	No built-in cancellation for async/await	Structured, built-in (Job.cancel())

Coroutines get the best of both worlds compared to what's likely familiar already: the readable, sequential code style of JavaScript's `async`/`await`, combined with real multi-core parallelism when needed (via `Dispatchers.Default`/`IO`) — something JavaScript's single-threaded model can't offer at all.

💻 Coding Challenges

Challenge 1: A Basic Suspend Function with launch

Write a suspend function `fun printAfterDelay(message: String, delayMs: Long)` that delays by `delayMs` then prints `message`. In `main()` (using `runBlocking`), use `launch` to start it with a 1000ms delay, print `"Launched"` immediately after, then use the returned `Job`'s `join()` so the program waits for it to finish before exiting.

Goal: Practice writing a suspend function and launching it with launch + join().

→ Solution

Challenge 2: Concurrent async Calls

Write two suspend functions, `fetchTemperature()` and `fetchHumidity()`, each delaying 500ms and returning an `Int`. In `main()`, use `async` to start both concurrently, then `await()` both results and print them together. Add a comment noting roughly how long the whole thing should take.

Goal: Practice concurrent async calls and see they run in parallel, not sequentially.

→ Solution

Challenge 3: Choosing a Dispatcher

Write a suspend function `fun simulateNetworkCall(): String` that runs on `Dispatchers.IO` (using `withContext`), delays 300ms, and returns a string. Call it from `main()` via `runBlocking` and print the result, along with a comment explaining why `Dispatchers.IO` is the appropriate choice here rather than `Dispatchers.Default`.

Goal: Practice withContext() and reason about which dispatcher fits a given kind of work.

→ Solution

💡 This Chapter Is the Foundation for the Rest of Course 2

suspend/launch/async/Dispatchers are the vocabulary every later Course 2 chapter assumes — Flow (next chapter) is built on top of suspend functions, and later chapters on generics, delegation, and DSLs all show up in real Kotlin code that's frequently coroutine-based. Getting comfortable with structured concurrency now pays off throughout the rest of the course.

🎯 What's Next

Next chapter: **Coroutines Advanced** — Flow, StateFlow, SharedFlow, structured concurrency in depth, and cancellation.

🌊 Coroutines Advanced

Chapter 1 covered a single asynchronous value at a time — one suspend call, one result. This chapter covers streams of values over time (Flow), the two flavors of "hot" stream built on it (StateFlow and SharedFlow), and how structured concurrency and cancellation actually behave once coroutines are nested and things go wrong.

🌀 Flow — Cold Asynchronous Streams

A `Flow<T>` represents a stream of values produced over time, asynchronously — the streaming counterpart to a single suspend function's one-shot result:

```
fun countdown(): Flow<Int> = flow {
    for (i in 3 downTo 1) {
        delay(1000)
        emit(i)  // produces one value into the stream
    }
}

fun main() = runBlocking {
    countdown().collect { value ->
        println(value)
    }
}
// Output (one per second): 3, 2, 1
```

`flow { }` builds a `Flow`; `emit()` produces a value into it; `collect { }` is how a consumer receives each value as it arrives. Crucially, `Flow` is **cold** — the code inside `flow { }` doesn't run at all until `collect` is called, and it runs *again from scratch* for every separate collector. Nothing happens just by calling `countdown()` — it only produces a description of the stream, not the stream itself.

vs Chapter 1's Sequence

A `Sequence` (Course 1, Chapter 6) is also lazy, but entirely synchronous — no suspending, no delays. `Flow` is the asynchronous counterpart: its builder block and its operators can freely call suspend functions like `delay()`.

Flow Operators Feel Familiar

`Flow` has `map`, `filter`, and similar operators with the exact same names as List/Sequence operators (Course 1, Chapter 6) — they just work value-by-value, over time, and are suspend-aware.

```
countdown()
    .map { it * 10 }
    .filter { it > 10 }
    .collect { println(it) }
// Output: 30, 20 — (1*10 = 10 is filtered out)
```

📌 StateFlow — Hot, Always Has a Value

`StateFlow` is a different shape entirely: it's **hot** (running independently of collectors) and **always holds a current value**, which every new collector receives immediately upon subscribing:

```
val counter = MutableStateFlow(0)  // requires an initial value

fun main() = runBlocking {
    launch {
        counter.collect { value ->
            println("Counter is now: $value")
        }
    }
    delay(100)
    counter.value = 1
    delay(100)
    counter.value = 2
}
// Output: Counter is now: 0
//         Counter is now: 1
//         Counter is now: 2
```

`counter.value` can be read or set directly at any time, from outside a coroutine even — `StateFlow` is the coroutine world's answer to "a variable that broadcasts its own changes." This is exactly the pattern used for UI state in modern Android apps with Jetpack Compose: a screen's entire visible state is typically one `StateFlow`, and the UI recomposes automatically whenever it changes.

📡 SharedFlow — Hot, Broadcast Events

`SharedFlow` is also hot, but has **no current value** — it broadcasts events to whoever happens to be collecting at the moment, and doesn't replay anything to a late subscriber by default:

```
val events = MutableSharedFlow<String>()  // no initial value required

fun main() = runBlocking {
    launch {
        events.collect { event ->
            println("Received: $event")
        }
    }
    delay(100)  // give the collector time to start
    events.emit("User logged in")
    events.emit("Item added to cart")
}
```

StateFlow vs SharedFlow

	StateFlow	SharedFlow
Has a current value?	Yes — always readable via .value	No
New collector gets...	The current value immediately	Nothing, until the next emit()
Best for	State (screen data, settings, counters)	One-off events (navigation, snackbar messages)
Duplicate values	Skipped if unchanged (by default)	Every emit() delivered

⚠ Don't Use StateFlow for One-Off Events

A common bug: modeling a "navigate to screen X" event as a `StateFlow`. Because `StateFlow` always holds its last value, a screen that re-subscribes later (e.g. after rotation on Android) immediately re-receives that old "navigate" event and fires it again unintentionally. `SharedFlow` — which doesn't replay by default — is the correct tool for genuine one-off events.

🪜 Structured Concurrency: coroutineScope vs supervisorScope

Chapter 1 introduced structured concurrency at a high level — child coroutines tied to their scope's lifetime. `coroutineScope` and `supervisorScope` both create a scope for launching children, but differ in how a child's failure affects its siblings:

```
// coroutineScope: one child failing cancels ALL siblings
coroutineScope {
    launch { delay(100); throw RuntimeException("Failed!") }
    launch { delay(500); println("Never printed — cancelled by sibling's failure") }
}

// supervisorScope: siblings are independent
supervisorScope {
    launch { delay(100); throw RuntimeException("Failed!") }
    launch { delay(500); println("Still printed — unaffected by sibling's failure") }
}
```

Use plain `coroutineScope` (the default, and what most nested async work should use) when a group of tasks are all part of one logical operation and a failure in any of them should abort the whole thing. Use `supervisorScope` when children are genuinely independent — e.g. several unrelated background tasks where one failing shouldn't take down the others.

🛑 Cancellation Is Cooperative

Calling `job.cancel()` doesn't forcibly kill a coroutine — it requests cancellation, and the coroutine itself has to cooperate by checking for it:

```
val job = launch {
    var i = 0
while (isActive) {  // checking isActive makes this loop cancellable
        println("Working... ${i++}")
        delay(200)  // delay() itself checks for cancellation and throws if cancelled
    }
}
delay(1000)
job.cancel()
```

Suspend functions like `delay()` already check for cancellation automatically and throw a `CancellationException` when it happens — that's why the loop above stops even without an explicit `isActive` check in simple cases. But a tight loop doing pure computation with **no** suspend calls inside it won't notice cancellation at all unless it explicitly checks `isActive` or calls `ensureActive()`.

⚠ Cleanup on Cancellation: try/finally + NonCancellable

A `finally` block still runs on cancellation, but any *suspend* call inside that `finally` block will itself immediately throw, since the coroutine is already cancelled by that point. If cleanup genuinely needs to suspend (e.g. an async "close connection" call), wrap just that part in `withContext(NonCancellable) { ... }` to let it complete despite the surrounding cancellation.

💻 Coding Challenges

Challenge 1: A Flow with Operators

Write a function `fun numberStream(): Flow<Int>` using the `flow { }` builder that emits the numbers 1 through 5, delaying 200ms before each emit. In `main()`, collect it after chaining `.filter { it % 2 == 0 }` and `.map { it * it }`, printing each resulting value.

Goal: Build and collect a Flow, chaining familiar operators onto an asynchronous stream.

→ Solution

Challenge 2: A StateFlow Counter

Create a `MutableStateFlow(0)` named `counter`. In `main()` (via `runBlocking`), launch a coroutine that collects `counter` and prints every value it sees. Then, from the main coroutine, update `counter.value` three times with a short delay between each update, and make sure the program waits long enough to see every printed update before exiting.

Goal: Practice reading/writing StateFlow.value and observing it from a separate coroutine.

→ Solution

Challenge 3: Cancelling a Running Coroutine

Launch a coroutine containing a `while (isActive)` loop that prints an incrementing counter every 300ms. From `main()`, let it run for about 1.5 seconds, then call `job.cancel()` and print `"Cancelled"` afterward. Confirm in a comment how many times you'd expect it to print before being cancelled.

Goal: Practice cooperative cancellation with isActive and job.cancel().

→ Solution

💡 Flow, StateFlow, and SharedFlow Cover Three Different Jobs

It's easy to reach for the wrong one. Ask: is this a finite (or ongoing) sequence of values to process (Flow)? Is it "the current state of something" that always has a value (StateFlow)? Or is it "something just happened, once" (SharedFlow)? Picking based on that question avoids most of the common misuse bugs — like the StateFlow-for-navigation-events trap covered above.

🎯 What's Next

Next chapter: **Generics & Variance** — `in`/`out`/invariant type parameters, reified types, type erasure, and generic constraints.

🧬 Generics & Variance

Generics let a class or function work with any type while staying type-safe — familiar territory if you've used Java generics or TypeScript's `<T>` syntax. This chapter covers the parts that go beyond the basics: type erasure, the `reified` keyword that works around it, and variance (`in`/`out`) — the part of generics most likely to be genuinely new.

📐 Generic Classes & Functions — A Quick Refresher

```
class Box<T>(val content: T)

fun <T> firstOrNull(list: List<T>): T? =
    if (list.isEmpty()) null else list[0]

val intBox = Box(42)        // Box<Int>, inferred
val stringBox = Box("hi")   // Box<String>, inferred
```

`<T>` is a placeholder type filled in at the call site — `Box<Int>` and `Box<String>` are different concrete types generated from the same generic definition, exactly as in Java or TypeScript. This part should already feel familiar; the rest of the chapter covers what's specific to Kotlin (and the JVM) about how generics actually behave.

🪆 Type Erasure

Like Java, Kotlin generics on the JVM are **erased** at runtime — the compiler enforces type safety while compiling, but the actual type parameter isn't available once the program is running:

```
fun <T> printType(value: T) {
    // println(value is T)    // Compile error: cannot check for erased type T
    println(value!!::class.simpleName)  // works — checks the RUNTIME class of the value itself
}
```

At runtime, a `List<String>` and a `List<Int>` are genuinely indistinguishable — both are just a `List`. This isn't a Kotlin limitation specifically; it's how the JVM itself represents generics, and it's the same reason Java has this restriction too.

⚠ vs TypeScript's Erasure

TypeScript also erases types (they're compile-time-only, gone entirely in the emitted JavaScript), but for a different reason — TypeScript's types are a pure compile-time overlay on dynamically-typed JS. Kotlin's erasure is a JVM platform constraint: the types are real and enforced by the compiler, they just aren't retained in the compiled bytecode's generic parameters.

🔓 reified — Working Around Erasure

Marking an `inline` function's type parameter `reified` makes the actual type available at runtime, because the function's body is copied directly into every call site at compile time — there's no erasure to fight, since the real type is known at each individual call:

```
inline fun <reified T> isType(value: Any?): Boolean {
    return value is T   // this compiles now — T is reified
}

println(isType<String>("hello"))  // true
println(isType<Int>("hello"))     // false
```

`reified` only works on `inline` functions — the compiler needs to literally paste the function body into each call site to substitute the real type in, which is only possible for inlined code. This is exactly how library functions like `filterIsInstance<T>()` work under the hood.

Java's Workaround

Java has no equivalent to `reified` — the standard workaround is passing an explicit `Class<T>` object as an extra parameter (`Gson`'s `fromJson(json, MyClass.class)` is a familiar example). Kotlin's `reified` avoids needing that extra parameter entirely.

Common Use: Type Checks & Casts

`reified` is mostly reached for when a generic function needs `is T`, `as T`, or `T::class` — anything that needs the actual runtime type, not just compile-time type safety.

↔️ Variance: in, out, and Invariant

By default, Kotlin generics are **invariant** — `Box<Dog>` and `Box<Animal>` are entirely unrelated types, even if `Dog` is a subtype of `Animal`:

```
open class Animal
class Dog : Animal()

class Box<T>(var content: T)

val dogBox: Box<Dog> = Box(Dog())
// val animalBox: Box<Animal> = dogBox  // Compile error: Box<Dog> is not a Box<Animal>
```

This is deliberately strict, and for good reason: if it were allowed, code could put a `Cat` (also an `Animal`) into what's really a `Box<Dog>` through the `Box<Animal>` reference, silently breaking type safety. `out` and `in` exist to safely relax this in the specific cases where it's actually sound.

```
// out T — covariant: a producer/read-only type. Producer<Dog> IS-A Producer<Animal>
interface Producer<out T> {
    fun produce(): T // T only ever comes OUT — safe to widen
}

val dogProducer: Producer<Dog> = object : Producer<Dog> {
    override fun produce() = Dog()
}
val animalProducer: Producer<Animal> = dogProducer  // fine — out makes this safe
// in T — contravariant: a consumer/write-only type. Consumer<Animal> IS-A Consumer<Dog>
interface Consumer<in T> {
    fun consume(item: T)   // T only ever goes IN — safe to narrow the other way
}

val animalConsumer: Consumer<Animal> = object : Consumer<Animal> {
    override fun consume(item: Animal) {}
}
val dogConsumer: Consumer<Dog> = animalConsumer  // fine — in makes this safe: anything that can handle a
ny Animal can handle a Dog
```

Variance Cheat Sheet

Modifier	Meaning	Type only appears…	Real example
(none) — invariant	No subtype relationship	In & out positions (read and write)	MutableList<T>
out — covariant	Producer<Dog> IS-A Producer<Animal>	Out (return values only)	List<out T> (Kotlin's List is read-only)
in — contravariant	Consumer<Animal> IS-A Consumer<Dog>	In (parameters only)	Comparator<in T>

This is exactly why Kotlin's read-only `List<T>` is declared `out` internally (it only ever produces `T`s, never accepts one) while `MutableList<T>` is invariant (it both produces and accepts `T`s via `add()`) — the compiler's variance rules directly mirror what's actually safe to do with the type.

🔒 Generic Constraints

A type parameter can be restricted to only accept types meeting some requirement, using an upper bound:

```
fun <T : Comparable<T>> max(a: T, b: T): T =
    if (a > b) a else b

println(max(3, 7))         // 7 — Int implements Comparable<Int>
println(max("apple", "banana"))  // banana — String implements Comparable<String>
// max(Dog(), Dog())    // Compile error: Dog doesn't implement Comparable<Dog>
```

`<T : Comparable<T>>` means "T can be anything, as long as it implements `Comparable<T>`" — this is what makes `a > b` valid inside the function body, since the compiler now knows every possible `T` supports comparison. Multiple constraints use a separate `where` clause:

```
fun <T> describe(item: T) where T : Comparable<T>, T : CharSequence {
    // T must satisfy BOTH constraints here
}
```

💻 Coding Challenges

Challenge 1: A Generic Constrained Function

Write a generic function `fun <T : Comparable<T>> middle(a: T, b: T, c: T): T` that returns the middle value of three (not the smallest, not the largest). Test it with three `Int`s and three `String`s.

Goal: Practice writing a generic function with an upper-bound constraint.

→ Solution

Challenge 2: A reified Type Check Function

Write an inline function `inline fun <reified T> countOfType(list: List<Any>): Int` that returns how many elements in `list` are of type `T`. Test it on a mixed list of `Int`s, `String`s, and `Double`s, counting each type separately.

Goal: Practice reified type parameters for a genuine runtime type check.

→ Solution

Challenge 3: Covariance with out

Write an `open class Animal(val name: String)` and a subclass `class Cat(name: String) : Animal(name)`. Write an interface `interface Shelter<out T : Animal>` with a function `fun adopt(): T`. Implement a `CatShelter : Shelter<Cat>`, then assign a `CatShelter` instance to a variable typed `Shelter<Animal>` and call `adopt()` through it.

Goal: See covariance (out) allow a more specific generic type to stand in for a more general one.

→ Solution

💡 out/in Read Like Their Names

A quick mnemonic: `out T` means T only flows *out* of the type (return values) — think "producer." `in T` means T only flows *in* (parameters) — think "consumer." If a type parameter is used both ways (as Course 1's mutable collections are), it has to stay invariant, since neither direction of subtyping is safe on its own.

🎯 What's Next

Next chapter: **Delegation** — `by lazy`, observable properties, map delegation, and writing custom delegates.

🎭 Delegation

Delegation lets a property (or a class) hand off its actual behavior to a separate helper object, using the `by` keyword. This chapter covers the built-in delegates you'll use constantly — `lazy`, `observable`, and map delegation — then shows how to write your own.

🔑 The by Keyword — What Delegation Means

A delegated property doesn't store its value directly — instead, getting and setting it is handed off to a separate **delegate** object, which implements the actual get/set logic:

```
val name: String by SomeDelegate()
//   ^ property       ^ the object that actually handles get()/set()
```

This is a single, general mechanism — `lazy`, `observable`, and map delegation (covered next) are all just pre-built delegate objects from the standard library, not separate language features. Once the pattern clicks, writing a custom delegate later in this chapter is a small step, not a new concept.

💤 by lazy — Lazy Initialization

`lazy` delays computing a property's value until the first time it's actually read, then caches the result for every read after that:

```
class Report {
    val data: String by lazy {
        println("Computing data...")  // only runs once, on first access
"Expensive result"
    }
}

val report = Report()
println("Report created")
println(report.data)   // "Computing data..." prints HERE, not at construction
println(report.data)   // no "Computing data..." — cached result reused
```

This is ideal for expensive setup work that a property might never actually need — parsing a large config file, opening a connection, building a lookup table — paying the cost only if and when it's genuinely used.

Thread Safety by Default

`lazy { }` defaults to `LazyThreadSafetyMode.SYNCHRONIZED` — safe if multiple threads might access the property simultaneously, at a small locking cost. Pass `lazy(LazyThreadSafetyMode.NONE) { }` to skip the lock when you know only one thread will ever touch it.

val Only — Not var

`lazy` only supports `val`, since the whole point is "compute once, then never change." A lazily-initialized *mutable* value uses a different delegate — `Delegates.notNull()` or a custom delegate — not `lazy`.

👀 Delegates.observable — Reacting to Changes

`Delegates.observable` runs a callback every time a `var` property is reassigned, receiving the old and new values:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("Unnamed") { property, old, new ->
        println("${property.name} changed from '$old' to '$new'")
    }
}

val user = User()
user.name = "Philip" // name changed from 'Unnamed' to 'Philip'
user.name = "Phil" // name changed from 'Philip' to 'Phil'
```

This is a compact way to get "whenever this changes, do X" without manually writing a custom setter — useful for logging, triggering a UI refresh, or keeping a derived value in sync. A close cousin, `Delegates.vetoable`, works the same way but its callback returns a `Boolean` deciding whether the change is allowed at all, rejecting it (keeping the old value) if it returns `false`.

🗺️ Map Delegation

Properties can be delegated directly to entries in a `Map`, keyed by the property's name — a compact way to represent an object backed by loosely-structured data:

```
class Config(map: Map<String, Any?>) {
    val host: String by map
    val port: Int by map
}

val config = Config(mapOf("host" to "localhost", "port" to 8080))
println(config.host)  // localhost — read from map["host"]
println(config.port)  // 8080 — read from map["port"]
```

Each property's name is used as the map key automatically — `val host: String by map` reads `map["host"]` under the hood. This is the exact mechanism used to build typed wrappers around loosely-structured data like a parsed JSON object or a set of raw config values, giving properties strong types on top of a plain `Map`. A `MutableMap` can be used the same way with `var` properties, writing back into the map on assignment.

🛠️ Writing a Custom Delegate

Any class can act as a property delegate, as long as it provides `getValue` (and, for a `var`, `setValue`) as **operator functions**:

```
import kotlin.reflect.KProperty

class LoggingDelegate<T>(private var value: T) {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): T {
        println("Reading ${property.name}: $value")
        return value
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, newValue: T) {
        println("Setting ${property.name} to $newValue")
        value = newValue
    }
}

class Settings {
    var volume: Int by LoggingDelegate(50)
}

val settings = Settings()
settings.volume = 75 // Setting volume to 75
println(settings.volume)  // Reading volume: 75 → 75
```

This is exactly what `lazy`, `Delegates.observable`, and map delegation already do internally — they're just pre-written classes with `getValue`/`setValue` operator functions, following this same shape. Writing a custom one makes sense whenever a piece of get/set behavior (validation, logging, caching, syncing to storage) needs to be reused across several properties or classes.

Kotlin Property Delegation vs Java

Behavior	Java's Approach	Kotlin
Lazy initialization	Manual null-check-then-compute in the getter, or a library	by lazy { ... }
React to a change	Hand-written setter with extra logic	by Delegates.observable(...) { }
Reusable get/set behavior	Composition + manual forwarding methods	A single custom delegate class, reused via by

⚠ Class Delegation Is a Different (Related) Feature

Kotlin also has **class delegation**, using the same `by` keyword on a class implementing an interface: `class Wrapper(base: Base) : Base by base` forwards every interface method to `base` automatically, only requiring overrides for methods that need different behavior. It shares the `by` keyword and the general "hand off to another object" idea with property delegation, but is a separate mechanism — worth knowing the name exists, even though this chapter's focus is property delegation.

💻 Coding Challenges

Challenge 1: Lazy Initialization

Write a class `DataLoader` with a property `val expensiveData: List<Int>` delegated with `by lazy { }`, which prints `"Loading data..."` before returning `listOf(1, 2, 3, 4, 5)`. In `main()`, create the loader, print a message confirming it was created, then access `expensiveData` twice — printed proof that the loading message only appears once.

Goal: Confirm lazy initialization runs once, on first access, not at construction.

→ Solution

Challenge 2: Observable Property

Write a class `ShoppingCart` with a `var itemCount: Int` delegated with `Delegates.observable(0) { ... }`, printing a message whenever it changes (old value → new value). Update `itemCount` three times in `main()` and confirm each change is logged.

Goal: Practice Delegates.observable for reacting to property changes.

→ Solution

Challenge 3: A Custom Range-Clamping Delegate

Write a custom delegate class `RangeClamped(private var value: Int, private val range: IntRange)` with `getValue`/`setValue` operator functions, where `setValue` clamps any assigned value into `range` before storing it (e.g. assigning 150 to a 0..100 range stores 100). Use it for a `var brightness: Int by RangeClamped(50, 0..100)` property and demonstrate it clamping an out-of-range assignment.

Goal: Write a genuinely useful custom delegate, not just log/print behavior.

→ Solution

💡 Recognize the Pattern, Not Just the Keywords

Every delegate in this chapter — built-in or custom — boils down to "something else owns getValue()/setValue()." Once that clicks, spotting when a custom delegate is worth writing becomes easy: any time the same non-trivial get/set logic (validation, clamping, logging, caching) would otherwise be copy-pasted across several properties.

🎯 What's Next

Next chapter: **DSL Building** — function literals with receiver, and building type-safe DSLs.

DSL Building

A DSL (Domain-Specific Language) is a mini-language for one specific job, built out of ordinary Kotlin — no separate parser, no new syntax to learn, just Kotlin code shaped to read like a description of what you want rather than a sequence of instructions. Gradle's Kotlin build scripts and Jetpack Compose's UI code are both real-world Kotlin DSLs. This chapter builds a small one from scratch, using the single feature that makes it possible: function literals with receiver.

🎯 The Feature Behind Every Kotlin DSL: Lambda with Receiver

A normal lambda parameter type looks like `(Int) -> String`. A **lambda with receiver** looks like `Int.() -> String` — inside the lambda body, `this` refers to that receiver, exactly as if the lambda's code were written as a method on that type:

```
// A regular lambda parameter — takes a StringBuilder as an argument
val normal: (StringBuilder) -> Unit = { sb -> sb.append("Hi") }

// A lambda WITH RECEIVER — StringBuilder becomes "this" inside the lambda
val withReceiver: StringBuilder.() -> Unit = { append("Hi") }  // no "sb." prefix needed
val sb = StringBuilder()
sb.withReceiver()  // called AS IF it were a method on sb
```

This is exactly how Kotlin's own `apply`, `with`, and `run` scope functions work — `obj.apply { doThing() }` reads naturally because the lambda passed to `apply` has `obj` as its receiver, so every member of `obj` is callable directly inside, without repeating `obj.` each time. DSL building is this same trick applied deliberately, on purpose-built types.

🧱 Building a Tiny HTML DSL, Step by Step

The goal: end up able to write something like this, and have it actually build a real HTML string:

```
val page = html {
    head { title("My Page") }
    body {
        h1("Welcome")
        p("Hello, world!")
    }
}
```

Step 1 — a base Tag class that knows how to hold children and render itself:

```
abstract class Tag(private val name: String) {
    private val children = mutableListOf<Tag>()
    private var text: String = ""
fun <T : Tag> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()        // runs the block WITH tag as the receiver
        children.add(tag)
        return tag
    }

    protected fun setText(value: String) { text = value }

    fun render(indent: String = ""): String {
        val body = if (text.isNotEmpty()) text else children.joinToString("") { "\n$it" }
        return "$indent<$name>$body</$name>"
    }

    override fun toString() = render()
}
```

Step 2 — specific tag classes and builder functions:

```
class Html : Tag("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)
    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : Tag("head") {
    fun title(text: String) = initTag(Tag("title") {}) { setText(text) }
}

class Body : Tag("body") {
    fun h1(text: String) = initTag(Tag("h1") {}) { setText(text) }
    fun p(text: String) = initTag(Tag("p") {}) { setText(text) }
}

fun html(init: Html.() -> Unit): Html {
    val h = Html()
    h.init()
    return h
}
```

Every builder function (`head { }`, `body { }`, and `html { }` itself) takes a lambda **with the relevant Tag subtype as its receiver**. That's the entire mechanism: inside `body { h1(...); p(...) }`, `h1` and `p` are directly callable because the lambda's implicit receiver is a `Body`, and `h1`/`p` are members of `Body`. No dot-prefixing, no explicit parameter name — it just reads like markup.

🔒 @DslMarker — Preventing Scope Leakage

Without any safeguard, an inner builder block can accidentally call a method from an *outer* receiver, since Kotlin allows reaching an outer implicit receiver when the inner one doesn't have a matching member:

```
html {
  body {
    head { title("Oops") } // without @DslMarker, this could accidentally compile —
                           // "head" isn't a Body member, but Kotlin might
                           // still resolve it against the outer Html receiver
  }
}
```

`@DslMarker` is an annotation you define once, then apply to the DSL's base type — it tells the compiler to only allow the *nearest* matching implicit receiver, blocking accidental calls to an outer one:

```
@DslMarker
annotation class HtmlDsl
@HtmlDsl
abstract class Tag(private val name: String) { /* ... */ }
```

With `@HtmlDsl` applied to `Tag` (and therefore inherited by `Html`, `Head`, `Body`, and everything else in this DSL), the compiler now enforces that inside `body { }`, only `Body`'s own members are reachable — the invalid `head { }` call above becomes a genuine compile error instead of a silent trap.

Building an Object: Java Builder Pattern vs Kotlin DSL

Java Builder Pattern	Kotlin DSL

`new PageBuilder()`
`  .title("My Page")`
`  .addHeading("Welcome")`
`  .addParagraph("Hello!")`
`  .build();`
	`html {`
	`  head { title("My Page") }`
	`  body {`
	`  h1("Welcome")`
	`  p("Hello!")`
	`  }`
	`}`

Both approaches build up a complex object step by step, but the DSL version expresses **structure and nesting** directly in the code's own shape — the indentation of `head { }` and `body { }` mirrors

the actual document tree, something a flat chain of `.method()` calls can't represent at all.

💻 Coding Challenges

Challenge 1: A Lambda-with-Receiver Warm-Up

Write a function `fun buildGreeting(init: StringBuilder.() -> Unit): String` that creates a `StringBuilder`, calls `init` on it as a receiver, and returns `.toString()`. Use it to build a greeting by calling `append(...)` three times inside the lambda with no receiver prefix.

Goal: Get comfortable with the T.() -> Unit syntax before building a full DSL.

→ Solution

Challenge 2: A Simple Menu DSL

Build a small DSL for describing a restaurant menu: a `Menu` class holding a list of item strings with a function `item(name: String, price: Double)` that formats and stores `"$name — $price"`, and a top-level `fun menu(init: Menu.() -> Unit): Menu`. Use it to build a 3-item menu and print each stored item.

Goal: Build a minimal DSL from scratch, following the html{} pattern from the chapter.

→ Solution

Challenge 3: Extending the HTML DSL

Using the chapter's `Tag`/`Html`/`Body` DSL as a starting point, add a new tag type `ul` (unordered list) to `Body`, which accepts a lambda that can call `li(text: String)` one or more times for list items. Build a small page using `body { ul { li("First") ; li("Second") } }` and print the rendered result.

Goal: Extend an existing DSL with a new nested builder, following the established pattern.

→ Solution

💡 You've Already Used This — Now You Know How It Works

Every time `apply { }`, `with(x) { }`, or a Gradle `dependencies { }` block has been written or read, it's been this exact mechanism in action. Building a DSL from scratch here isn't introducing something new so much as revealing the machinery behind patterns that already felt natural.

🎯 What's Next

Next chapter: **Reflection & Annotations** — `KClass`, the reflection API, custom annotations, and use-site targets.

🔬 Reflection & Annotations

Reflection is code that inspects other code's structure at runtime — class names, properties, functions — rather than working with fixed, known-in-advance types. Annotations attach extra metadata to declarations, which reflection can then read back. This chapter covers Kotlin's `KClass` reflection API, writing custom annotations, and the use-site targets Kotlin needs because a single property can compile into several different underlying elements.

🏷️ KClass — Kotlin's Reflection Entry Point

Every type has a corresponding `KClass` object, accessed with `::class`, which describes that type's structure:

```
data class Person(val name: String, val age: Int)

val kClass = Person::class
println(kClass.simpleName)      // Person
println(kClass.qualifiedName)   // (package).Person
val p = Person("Alice", 30)
println(p::class.simpleName)    // Person — from an instance, not just the type
```

`Type::class` works on a type name directly; `instance::class` works on a value, returning its actual runtime class — useful when a value is typed more generally than its concrete class (e.g. a variable typed `Animal` holding a `Dog`, as in Course 1's inheritance chapter).

⚠ kotlin-reflect Is a Separate Dependency

Basic `::class` and simple properties like `simpleName` work out of the box, but deeper reflection — listing a class's properties and functions, calling a function dynamically — requires the **kotlin-reflect** library as an explicit Gradle dependency. It's deliberately not bundled by default, since full reflection support adds real size to a compiled app, and most Kotlin code (especially on Android) never needs it.

🔎 Inspecting Members

With `kotlin-reflect` available, a `KClass` can list its own properties and functions:

```
import kotlin.reflect.full.memberProperties

val p = Person("Alice", 30)

for (prop in p::class.memberProperties) {
    println("${prop.name} = ${prop.get(p)}")
}
// name = Alice
// age = 30
```

`prop.get(p)` reads that property's value off a specific instance, generically — the code above works for *any* data class passed in, without knowing its properties in advance. This generic, "works on any type" capability is exactly what makes reflection useful for building libraries: a JSON serializer, for instance, can walk any object's properties this way and turn them into JSON fields without the library author needing to know about `Person` specifically.

🏷️ Custom Annotations

An annotation is declared with the `annotation` keyword, and can be applied to classes, properties, functions, or parameters:

```
annotation class Important(val reason: String)

@Important("Core domain model")
class Order(val id: Int)
```

By itself, an annotation is inert — it's just metadata attached to a declaration. It becomes useful once something reads it back, which is exactly what reflection is for:

```
val annotation = Order::class.annotations
    .filterIsInstance<Important>()
    .firstOrNull()

println(annotation?.reason)   // Core domain model
```

@Retention — How Long It Survives

`@Retention(AnnotationRetention.RUNTIME)` keeps an annotation available for reflection at runtime (the default). `SOURCE` discards it after compiling; `BINARY` keeps it in the compiled file but not visible to runtime reflection.

@Target — Where It Can Be Used

`@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)` restricts an annotation to only compile onto the listed kinds of declaration, catching misuse (like annotating a variable that was meant for classes only) at compile time.

🎯 Use-Site Targets — Why Kotlin Needs Them

A single Kotlin `val`/`var` property can compile down into *several* distinct underlying JVM elements at once — a backing field, a getter method, a setter method (for `var`), and sometimes a constructor parameter. An annotation applied to the property alone is ambiguous about which of those it should attach to:

```
class Config(
    @field:JvmField // applies to the underlying FIELD specifically
val debugMode: Boolean,

    @get:JvmName("getUserId")  // applies to the GETTER specifically
val userId: Int
)
```

Use-Site Target Prefixes

Prefix	Targets
@property:	The Kotlin property itself (the default when none is specified, where unambiguous)
@field:	The underlying backing field
@get:	The generated getter function
@set:	The generated setter function (var only)
@param:	The constructor parameter
@setparam:	The setter's parameter specifically

This is purely a Kotlin-specific concern — it exists because Kotlin properties are a higher-level abstraction over what Java expresses as separate field + getter + setter methods. Annotations that are Java-interop-focused (like `@JvmField` or Jackson/Gson's JSON annotations) very often need a specific use-site target to land on the correct underlying element.

⚠ Ambiguity Errors Are the Compiler Helping You

If an annotation could reasonably apply to more than one target and none is specified, Kotlin raises a compile error asking for a use-site target rather than silently guessing. This is a safety feature, not friction — a wrongly-placed annotation (e.g. a JSON library annotation that ends up on the field instead of the getter) can fail silently at runtime instead of at compile time if the compiler didn't insist on clarity here.

💻 Coding Challenges

Challenge 1: Inspecting a Class with KClass

Using a data class `Product(val name: String, val price: Double, val inStock: Boolean)`, write a function `fun describe(obj: Any)` that uses reflection (`obj::class.memberProperties`) to print every property name and value on the given object, one per line, in the format `name: value`.

Goal: Practice reading an object's properties generically via reflection, without hardcoding property names.

→ Solution

Challenge 2: A Custom Annotation with a Reader

Declare an annotation `@Target(AnnotationTarget.CLASS) annotation class Author(val name: String)` with `RUNTIME` retention. Apply it to a class with a made-up author name, then write a function `fun printAuthor(kClass: KClass<*>)` that finds the `Author` annotation on the given class (if present) and prints its name, or `"No author"` if absent.

Goal: Practice declaring, applying, and reading back a custom annotation via reflection.

→ Solution

Challenge 3: Use-Site Targets

Given the annotations `@Target(AnnotationTarget.FIELD) annotation class Sensitive` and `@Target(AnnotationTarget.PROPERTY_GETTER) annotation class Logged`, write a class with a property that applies `@Sensitive` to its backing field and `@Logged` to its getter, using the correct use-site target prefix for each. Add a comment explaining why a use-site target is required for each one.

Goal: Practice choosing the correct use-site target prefix based on an annotation's declared @Target.

→ Solution

💡 Reflection Is a Library-Building Tool, Not an Everyday One

Day-to-day application code rarely reaches for reflection directly — it's the mechanism *underneath* tools that feel automatic: JSON libraries mapping objects to fields, dependency injection frameworks wiring up constructors, testing frameworks discovering `@Test`-annotated functions. Understanding it here mostly pays off by demystifying how those tools work, not as something to reach for constantly in application code.

🎯 What's Next

Next chapter: **Advanced Patterns** — value classes, operator overloading, contracts, and type aliases.

⚙️ Advanced Patterns

This chapter is a grab-bag of four smaller, self-contained tools that come up throughout real Kotlin code: value classes for zero-cost type-safe wrappers, operator overloading for expressive custom types, contracts for teaching the compiler about your own smart-cast-friendly functions, and type aliases for readability.

📦 Value Classes — Type Safety with Zero Overhead

A common bug: a function taking two `Int` parameters (say, `createUser(id: Int, age: Int)`) is easy to call with the arguments accidentally swapped — the compiler can't catch it, since both are just `Int`. A `value class` wraps a single value in a distinct type that the compiler *can* tell apart, while compiling away to nothing extra at runtime:

```
@JvmInline
value class UserId(val value: Int)

@JvmInline
value class Age(val value: Int)

fun createUser(id: UserId, age: Age) { /* ... */ }

createUser(UserId(42), Age(30))
// createUser(Age(30), UserId(42))   // Compile error — wrong types, caught immediately
```

At runtime, a `UserId` is represented as a plain `Int` wherever possible — the wrapper is erased by the compiler in most cases, so there's no extra object allocation or memory overhead compared to using a raw `Int`. The type safety is entirely a compile-time guarantee, at effectively zero runtime cost.

⚠ Exactly One Property, Read-Only

A value class must wrap exactly one `val` property, and can't have additional state — this single-property restriction is what makes the "erase to the underlying type" optimization possible. It can still have functions and computed properties, just no extra stored state beyond the one wrapped value.

➕ Operator Overloading

Marking a function `operator` lets it be called using an operator symbol instead of its name — Kotlin maps specific function names (`plus`, `minus`, `times`, `compareTo`, and others) onto `+`, `-`, `*`, comparison operators, and more:

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point) = Point(x + other.x, y + other.y)
    operator fun times(scale: Int) = Point(x * scale, y * scale)
}

val p1 = Point(1, 2)
val p2 = Point(3, 4)
println(p1 + p2)   // Point(x=4, y=6) — calls p1.plus(p2)
println(p1 * 3)    // Point(x=3, y=6) — calls p1.times(3)
```

Common Overloadable Operators

Operator	Function Name	Operator	Function Name
a + b	plus	a[i]	get
a - b	minus	a[i] = v	set
a * b	times	a in b	contains
a > b / a < b	compareTo	a()	invoke
a == b	equals	-a	unaryMinus

This is the same underlying idea as Java's lack of true operator overloading (Java only special-cases `+` for `String` concatenation) versus, say, Python's `__add__` dunder methods — Kotlin's version is Python-like in spirit, but statically type-checked, and limited to a fixed, well-known set of operator names rather than allowing arbitrary custom syntax.

⚠ Use Sparingly — Readability Matters

Operator overloading is genuinely useful for types where the operation has an obvious, unsurprising meaning (vectors, money, matrices). Overloading `+` on a type where it isn't intuitively "addition" (making `user1 + user2` merge two accounts, say) tends to make code harder to read, not easier — the operator symbol should match what a reader would already guess it does.

📜 Contracts — Teaching the Compiler About Your Functions

Course 1's smart casts (Chapter 3) automatically narrow a nullable type after a built-in null check like `if (x != null)`. A `contract` block extends that same smart-cast behavior to a *custom* function:

```
import kotlin.contracts.contract

fun isNotNullOrEmpty(str: String?): Boolean {
    contract {
        returns(true) implies (str != null)
    }
    return !str.isNullOrEmpty()
}

fun greet(name: String?) {
    if (isNotNullOrEmpty(name)) {
        println(name.length)  // smart-cast to String — the compiler trusts the contract
    }
}
```

Without the `contract` block, `name.length` above wouldn't compile — the compiler has no way to know that `isNotNullOrEmpty` returning `true` guarantees `name` isn't null, since it's just an ordinary function as far as the compiler is concerned. The contract explicitly promises that relationship.

⚠ An Unchecked Promise — Get It Right

Contracts are **not verified** by the compiler — writing a contract that doesn't actually match the function's real behavior compiles fine, but produces incorrect smart-casts that can crash at runtime. This is genuinely advanced, low-level territory: most Kotlin code never writes a contract, and the standard library's own functions like `isNullOrEmpty` and `requireNotNull` are really the main examples worth knowing exist.

🏷️ Type Aliases — A Name, Not a New Type

`typealias` gives an existing type a new, more readable name — it's purely a compile-time label, not a distinct type the way a value class is:

```
typealias UserMap = Map<String, List<Int>>
typealias ClickHandler = (x: Int, y: Int) -> Unit
fun processUsers(users: UserMap) { /* ... */ }   // far more readable than the raw Map<...> type
val onClick: ClickHandler = { x, y -> println("Clicked at $x, $y") }
```

`UserMap` and `Map<String, List<Int>>` are **completely interchangeable** to the compiler — a `UserMap` can be passed anywhere a `Map<String, List<Int>>` is expected and vice versa, with no conversion. That interchangeability is exactly the difference from a value class: `typealias` is for readability on complex generic types, not for type safety.

value class vs typealias

	value class	typealias
Distinct type?	Yes — compiler tells it apart from the wrapped type	No — fully interchangeable with the original
Purpose	Type safety (prevent mixing up similar values)	Readability (shorten a long/complex type)
Runtime representation	Erased to the wrapped type where possible	N/A — never existed as a separate type

💻 Coding Challenges

Challenge 1: A Value Class for Type Safety

Define two value classes, `Meters(val value: Double)` and `Feet(val value: Double)`. Write a function `fun metersToFeet(m: Meters): Feet` that converts (1 meter = 3.28084 feet). Demonstrate calling it correctly, and add a commented-out line showing a call that would be a compile error (e.g. passing a `Feet` where `Meters` is expected).

Goal: Practice value classes preventing a realistic unit-mixup bug.

→ Solution

Challenge 2: Operator Overloading for a Money Type

Write a data class `Money(val cents: Int)` with overloaded `plus` (adding two `Money` values) and `compareTo` (comparing by `cents`) operators. Demonstrate adding two `Money` values with `+` and comparing two with `>`.

Goal: Practice writing operator functions with a genuinely intuitive use case.

→ Solution

Challenge 3: A Type Alias for a Callback

Define a type alias `ValidationResult` for `Pair<Boolean, String>` (valid flag + message), and a type alias `Validator` for `(String) -> ValidationResult`. Write a function `fun validateUsername(): Validator` returning a lambda that checks a username is at least 3 characters, returning an appropriate `ValidationResult`. Call it and print the result.

Goal: Practice using type aliases to make a function-type signature more readable.

→ Solution

💡 Four Tools for Four Different Jobs

Reach for a value class when two things share a primitive type but must never be mixed up. Reach for operator overloading when an operation on a custom type has an obvious real-world meaning. Reach for a type alias purely to make a complex type easier to read. Contracts are the outlier — worth understanding, rarely worth writing yourself.

🎯 What's Next

Next chapter — the final chapter of Kotlin Intermediate: **Kotlin Multiplatform Basics** — KMP intro, the expect/actual pattern, and sharing code across platforms.

🌍 Kotlin Multiplatform Basics

The final chapter of this course is the payoff for a language built without primitives, with erasure that's a JVM-specific concern rather than a language one: Kotlin Multiplatform (KMP) lets one Kotlin codebase target the JVM/Android, iOS, JavaScript, and native desktop from shared source. This chapter covers what KMP actually is, project structure, and the expect/actual pattern that makes platform-specific code possible within a shared codebase.

🧩 What Kotlin Multiplatform Actually Shares

KMP's core idea: write business logic, data models, and networking **once**, in plain Kotlin, and compile that same code to run natively on every target platform — not through a bridge or interpreter, but as genuinely native code on each platform (JVM bytecode for Android, native binaries for iOS via Kotlin/Native, JavaScript for web).

Typically Shared

Data models, business logic, ViewModels, networking code, local database access, validation rules — anything that's pure logic without a platform-specific UI dependency.

Typically Platform-Specific

The UI layer — Jetpack Compose on Android, SwiftUI on iOS — stays native to each platform in the traditional KMP approach, so each platform still looks and feels genuinely native.

⚠ Not the Same Approach as React Native or Flutter

React Native and Flutter share the UI layer itself across platforms (rendering through their own cross-platform UI toolkit). Traditional KMP deliberately does the opposite — share the logic, keep the UI 100% native per platform. (Compose Multiplatform, a newer, separate JetBrains project built on KMP, does extend sharing to the UI layer too — worth knowing it exists, but it's a distinct, additional layer on top of what this chapter covers.)

📁 Project Structure — Source Sets

A KMP project splits code into **source sets** — folders that each target a different scope of platforms:

```
project/
├── commonMain/      # shared Kotlin code — compiles for EVERY target
│   └── kotlin/
│       └── Repository.kt
├── androidMain/     # Android-only code
│   └── kotlin/
│       └── PlatformInfo.kt
├── iosMain/         # iOS-only code
│   └── kotlin/
│       └── PlatformInfo.kt
└── jvmMain/         # desktop JVM-only code (if targeted)
    └── kotlin/
        └── PlatformInfo.kt
```

`commonMain` is where the bulk of shared logic lives, and it's compiled separately for each actual target platform — the same source file produces genuinely different compiled output per platform (JVM bytecode, a native iOS binary, JS), even though it's written once.

🔀 expect / actual — Platform-Specific Implementations

Shared code in `commonMain` sometimes needs something only a specific platform can actually provide (a device ID, a platform name, a native API). `expect` declares the shape of that thing in shared code, without an implementation; each platform then supplies its own `actual`:

```
// commonMain/kotlin/Platform.kt — declares WHAT exists, not HOW
expect fun platformName(): String
// commonMain — this can use platformName() without knowing which platform it's on
fun greet(): String = "Hello from ${platformName()}!"

// androidMain/kotlin/Platform.kt
actual fun platformName(): String = "Android"
// iosMain/kotlin/Platform.kt
actual fun platformName(): String = "iOS"
```

`greet()`, written once in `commonMain`, produces `"Hello from Android!"` when compiled for Android and `"Hello from iOS!"` when compiled for iOS — the shared code calls `platformName()` without ever knowing (or needing to know) which platform's `actual` will end up wired in. Each platform's compiler substitutes its own `actual` for every `expect` declaration at compile time.

expect/actual Works on More Than Functions

Classes, properties, and objects can all be declared `expect` in common code with an `actual` per platform — not just top-level functions. A common pattern: an `expect class DatabaseDriver` with a platform-specific SQLite driver as each `actual`.

Every Platform Must Supply an actual

If a target platform is missing an `actual` for some `expect` declaration, the build fails for that platform specifically — the compiler enforces that every declared expectation is genuinely met everywhere it's compiled.

📦 Where Third-Party Libraries Fit In

Networking, serialization, and database libraries increasingly ship KMP-aware versions (Ktor for networking, kotlinx.serialization for JSON, SQLDelight for databases) — these already handle their own `expect`/`actual` internally, so shared code calling them doesn't need to write any platform-specific code itself. Reaching for an `expect`/`actual` pair by hand is really a last resort, for the genuinely platform-specific slice of an app that no shared library already covers.

Cross-Platform Approaches, Compared

Approach	What's Shared	UI
Kotlin Multiplatform	Business logic, data, networking	Native per platform (Compose / SwiftUI)
Compose Multiplatform	Logic + UI	Shared Compose UI across platforms
React Native	Logic + UI	Shared UI, rendered via native components
Flutter	Logic + UI	Shared UI, rendered via Flutter's own engine

💻 Coding Challenges

Challenge 1: A Basic expect/actual Pair

Write an `expect fun currentTimestamp(): Long` in "commonMain" (write it as a single file, with a comment marking where it belongs), then two `actual` implementations in comments labeled "androidMain" and "jvmMain" — both can call `System.currentTimeMillis()`, but write them as if in separate files/platforms to show the pattern. Add a shared function `fun describeTimestamp(): String` that uses `currentTimestamp()`.

Goal: Practice the expect/actual declaration pattern, even without a real multi-target build.

→ Solution

Challenge 2: An expect class

Write an `expect class Logger` with a single function `fun log(message: String)` in "commonMain" (as a comment-labeled section), then two "actual class Logger" implementations (labeled "androidMain" and "iosMain" in comments) — one prefixing messages with `"[Android]"`, the other with `"[iOS]"`. Write a shared function that takes a `Logger` and logs a test message.

Goal: Practice expect/actual on a class, not just a function.

→ Solution

Challenge 3: Designing a Shared/Platform Split

For a hypothetical note-taking app shared across Android and iOS via KMP, list (in comments, as a short design doc) which of the following belong in commonMain vs need an expect/actual split vs stay fully platform-specific: note data model, note validation logic, local file storage, the note list UI screen, network sync logic, push notification handling.

Goal: Practice reasoning about what's genuinely shareable in a KMP app, not just writing expect/actual syntax.

→ Solution

💡 Course 2 Complete — Straight Into Android Territory

Kotlin Intermediate closes here having covered coroutines, generics, delegation, DSLs, reflection, advanced patterns, and now multiplatform — genuinely everything needed to read and write real-world Kotlin, Android or otherwise. The planned Android Development courses build directly on this: Jetpack Compose leans on lambdas-with-receiver (DSL building) and sealed classes (Course 1) for UI state, ViewModels lean on coroutines and StateFlow, and KMP is directly relevant if a shared/cross-platform app is ever the goal.

🎯 What's Next

Kotlin Intermediate (Course 2) is complete. From here: Android Development Course 1 (Fundamentals) picks up directly where this leaves off, applying everything from both Kotlin courses to real Android Studio projects.