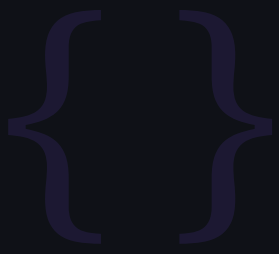

KOTLIN

Fundamentals

Getting Started · Functions & Lambdas · Null Safety

Classes & Data Classes · Inheritance & Interfaces

Collections & Functional Operators · Control Flow & Destructuring

Extension Functions & Object Declarations

Philip Osztromok

Generated with Claude

Table of Contents

Kotlin Fundamentals — 8 Chapters

🚀 Getting Started with Kotlin

Kotlin is a modern, statically-typed language created by JetBrains that runs on the JVM (the same virtual machine Java runs on) and has been Android's officially preferred language since 2019. This chapter covers what Kotlin is and why it exists, how it compares to Java and JavaScript, and the absolute basics — declaring values, picking a type, and building strings.

What Is Kotlin, and Why Learn It?

Kotlin is:

- **Statically typed** — every value has a type, checked at compile time (like TypeScript, unlike plain JavaScript)
- **JVM-based** — Kotlin source compiles to the same bytecode Java does, and runs on the same virtual machine
- **100% Java-interoperable** — Kotlin code can call Java libraries directly, and vice versa
- **Android's preferred language** — Google made Kotlin the recommended language for Android development in 2019, ahead of Java
- **Designed to fix Java's pain points** — null safety, less boilerplate, and expression-oriented syntax are all direct responses to common Java frustrations

Kotlin vs Java vs JavaScript, at a Glance

Trait	JavaScript	Java	Kotlin
Typing	Dynamic	Static, verbose	Static, inferred
Where it runs	Browser / Node	JVM	JVM (also JS, Native)
Null handling	undefined / null, unchecked	null, unchecked (NPEs)	Null safety built into the type system
Semicolons	Optional (ASI)	Required	Optional
Entry point	Top-level script code	public static void main(String[] args)	fun main()

▶️ Your First Kotlin Program

Hello, Kotlin

```
// hello.kt
fun main() {
    println("Hello, Kotlin!")
}
```

Run it (from the command line, with the Kotlin compiler installed):

```
$ kotlinc hello.kt -include-runtime -d hello.jar
$ java -jar hello.jar
Hello, Kotlin!
```

In practice, almost nobody runs Kotlin this way day to day — IntelliJ IDEA (or Android Studio, which is built on it) compiles and runs Kotlin with a single click. The command-line path above is worth seeing once so `fun main()` doesn't feel like magic: it compiles to a `.jar`, which runs on the JVM exactly like compiled Java would.

Notice there's no class wrapping `main`, and no semicolon at the end of the `println` line — both required in Java, both optional in Kotlin. `fun` declares a function (short for *function*), and top-level functions — ones that live outside any class — are completely normal in Kotlin, unlike Java where every method must belong to a class.

📦 val vs var

Kotlin has exactly two ways to declare a value, and the choice is deliberate every time:

```
val name = "Philip" // read-only reference — cannot be reassigned
var score = 0 // mutable — can be reassigned

score = 10 // fine, score is a var
name = "Someone else" // compile error — val cannot be reassigned
```

val — like JS const

A `val` is a read-only reference, assigned once. It maps directly onto JavaScript's `const` — same idea, same restriction. In Kotlin, `val` is the default choice; you reach for `var` only when you know a value needs to change.

var — like JS let

A `var` can be reassigned, the same as JavaScript's `let`. There's no Kotlin equivalent of JavaScript's old `var` keyword with its confusing function-scoping — Kotlin's `var` behaves like a normal block-scoped mutable variable.

Declaring a Mutable Value: Java vs Kotlin

Language	Immutable	Mutable
Java	`final int score = 0;`	`int score = 0;`
JavaScript	`const score = 0;`	`let score = 0;`
Kotlin	`val score = 0`	`var score = 0`

Java's default is mutable — you have to opt in to immutability with `final`, and most Java code doesn't bother. Kotlin flips that: `val` is the natural first choice, which pushes code toward fewer accidental mutations without any extra ceremony.

🔢 Basic Types & Type Inference

Kotlin's built-in types cover the same ground as Java's primitives, but without the primitive/object split:

```
val age: Int = 33
val price: Double = 19.99
val isReady: Boolean = true
val initial: Char = 'P'
val title: String = "Learning Blog"
// The type annotations above are optional — Kotlin infers them:
val inferredAge = 33 // inferred as Int
val inferredPrice = 19.99 // inferred as Double
```

vs Java: No Primitives

Java has a hard split between primitives (`int`, `double`, `boolean`) and their boxed object wrappers (`Integer`, `Double`, `Boolean`). Kotlin has just one type per concept — `Int`, `Double`, `Boolean` — and the compiler quietly uses the efficient primitive form under the hood wherever it can.

vs JavaScript: No Number-for-Everything

JavaScript has a single `number` type for all numeric values. Kotlin distinguishes `Int` (whole numbers) from `Double` (decimals) at compile time — mixing them without an explicit conversion is a compile error, not a silent runtime surprise.

Type inference means you rarely write the type annotation explicitly — Kotlin figures out `Int`, `String`, etc. from the value on the right-hand side, the same way TypeScript's `let x = 5` infers `number`. The annotation is still useful (and sometimes required) when the type isn't obvious from context, such as an empty collection or a function parameter.

📝 String Templates

Kotlin builds strings with **string templates** — the direct equivalent of JavaScript's template literals, but with a different symbol:

```
val name = "Philip"
val age = 33
// Simple variable — just $name
println("Hello, $name!")              // Hello, Philip!
// Expression — needs ${ } braces
println("Next year you'll be ${age + 1}") // Next year you'll be 34
```

Template Syntax: JavaScript vs Kotlin

	JavaScript	Kotlin
Delimiter	Backticks: `...`	Double quotes: "..."
Simple variable	${name}	$name
Expression	${age + 1}	${age + 1}

The key difference: Kotlin string templates live inside ordinary double-quoted strings, not a special delimiter — so every `String` in Kotlin is automatically template-capable. For a single variable, the braces are optional (`$name` works fine); for anything more than a bare variable — a property access, a function call, an expression — the `${ }` braces are required.

⚠ $ needs escaping if you mean it literally

Because `$` triggers template interpolation, a literal dollar sign in a Kotlin string needs escaping: `"Price: \$19.99"`. This is the same category of gotcha as needing to escape a literal backtick inside a JavaScript template literal.

💻 Coding Challenges

Challenge 1: val vs var in Practice

Write a Kotlin program with a `val` holding your name and a `var` holding a counter starting at 0. Increment the counter three times and print it after each increment. Then try (and comment out) reassigning the `val` — note the compiler error.

Goal: Build muscle memory for choosing `val` by default and reaching for `var` only when needed.

→ Solution

Challenge 2: Typed Values & Inference

Declare five `val`s — one each of type `Int`, `Double`, `Boolean`, `Char`, and `String` — first with explicit type annotations, then rewritten with type inference (no annotations). Print all five using one `println` per value.

Goal: Get comfortable with Kotlin's five basic types and when inference does the work for you.

→ Solution

Challenge 3: String Templates

Given a `val` for a product name and a `var` for its price (a `Double`), print a single sentence using a string template that includes the name (simple interpolation) and the price with 10% tax added (expression interpolation with `${ }`).

Goal: Practice both forms of string template interpolation in one realistic sentence.

→ Solution

💡 Why This Course Starts on the JVM Track

Kotlin also compiles to JavaScript (Kotlin/JS) and native binaries (Kotlin/Native), but this course sticks to the JVM track throughout — it's the version used for Android development, which is the reason Kotlin is on the syllabus in the first place. Everything here transfers directly once the Android courses begin.

🎯 What's Next

Next chapter: **Functions & Lambdas** — default parameters, named arguments, single-expression functions, and Kotlin's lambda syntax.

🔧 Functions & Lambdas

Functions are where Kotlin's "less boilerplate than Java, more structure than JavaScript" philosophy shows up clearly. This chapter covers declaring functions, giving parameters default values, calling with named arguments, writing single-expression functions, and Kotlin's lambda syntax — the foundation for the functional-style APIs (like collection operators) used constantly later in this course.

📐 Declaring a Function

```
fun greet(name: String): String {
    return "Hello, $name!"
}

println(greet("Philip"))   // Hello, Philip!
```

Every parameter must have an explicit type — Kotlin does not infer parameter types the way it infers `val` types, since the compiler has no value to look at until the function is called. The return type (`: String` above) can often be omitted and inferred, but many teams write it explicitly on public functions for readability.

Function Syntax: Java vs JavaScript vs Kotlin

Language	Declaring greet(name)
Java	public String greet(String name) { return "Hello, " + name; }
JavaScript	function greet(name) { return `Hello, ${name}!`; }
Kotlin	fun greet(name: String): String { return "Hello, $name!" }

➡️ Single-Expression Functions

When a function body is just a single `return` expression, Kotlin lets you drop the braces and the `return` keyword entirely, using `=` instead:

```
// Long form
fun square(x: Int): Int {
    return x * x
}

// Single-expression form — equivalent
fun square(x: Int) = x * x
```

The return type is inferred from the expression (`Int * Int` is `Int`), so it's usually left off in single-expression form. This isn't just shorthand for its own sake — Kotlin code leans heavily on small, expression-bodied functions, especially once lambdas and collection operators enter the picture later in the course.

Default Parameters

A parameter can have a default value, making the argument optional at the call site: `fun greet(name: String, greeting: String = "Hello")`. Calling `greet("Philip")` uses the default; `greet("Philip", "Hi")` overrides it.

Named Arguments

Any call can name its arguments explicitly: `greet(name = "Philip", greeting = "Hi")`. This lets you skip earlier defaults and go straight to a later one, or just make a call with many parameters self-documenting.

```
fun createUser(name: String, age: Int = 18, isAdmin: Boolean = false) {
    println("$name, age $age, admin=$isAdmin")
}

createUser("Alice")                    // Alice, age 18, admin=false
createUser("Bob", 25)                  // Bob, age 25, admin=false
createUser("Carol", isAdmin = true)    // Carol, age 18, admin=true — skips age via named arg
```

⚠ No Function Overloading Juggling Needed

In Java, "optional" parameters are usually simulated with several overloaded method signatures (`createUser(name)`, `createUser(name, age)`, `createUser(name, age, isAdmin)`) — three methods to maintain in sync. Kotlin's default parameters replace all of that with one function.

🎯 Lambda Basics

A lambda is a function without a name, written inline and often passed as an argument — conceptually identical to a JavaScript arrow function:

```
// Kotlin lambda, stored in a val
val square = { x: Int -> x * x }
println(square(5))  // 25
// Passed directly as an argument, e.g. to a collection operator
val numbers = listOf(1, 2, 3, 4)
val doubled = numbers.map { n -> n * 2 }
println(doubled)  // [2, 4, 6, 8]
// Single parameter can use the implicit "it"
val tripled = numbers.map { it * 3 }
println(tripled)  // [3, 6, 9, 12]
```

Lambda Syntax: JavaScript vs Kotlin

	JavaScript	Kotlin
Arrow / lambda	(x) => x * x	{ x: Int -> x * x }
Separator	=>	-> (inside braces)
Implicit single param	Not available	it
Trailing lambda call	arr.map((x) => x * 2)	list.map { it * 2 }

That last row matters: when a lambda is the *last* argument to a function — which it is for almost every collection operator (`map`, `filter`, `forEach`…) — Kotlin lets you move it outside the parentheses entirely, and drop the (now-empty) parentheses altogether. `numbers.map({ it * 2 })` and `numbers.map { it * 2 }` are the same call; the second is idiomatic Kotlin.

💻 Coding Challenges

Challenge 1: Default Parameters & Named Arguments

Write a function `orderPizza(size: String = "medium", toppings: Int = 1, extraCheese: Boolean = false)` that prints an order summary. Call it four times: with no arguments, with only `size` overridden, with only `extraCheese` overridden using a named argument, and with all three provided.

Goal: Get comfortable mixing default values with named-argument overrides.

→ Solution

Challenge 2: Single-Expression Functions

Write three functions as single-expression functions (using `=`, no braces or `return`): `isEven(n: Int)` returning a `Boolean`, `max(a: Int, b: Int)` returning the larger value, and `greeting(name: String)` returning a greeting string using a string template.

Goal: Practice recognizing when a function body is simple enough to collapse to a single expression.

→ Solution

Challenge 3: Lambdas with map and filter

Given `val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`, use a lambda with `filter` to keep only even numbers, then use a lambda with `map` to square each of the results. Print the final list. Write the lambdas using the implicit `it` parameter.

Goal: Chain two lambda-based collection operators and get comfortable with `it`.

→ Solution

💡 Trailing Lambda Syntax Is Everywhere

The "lambda moved outside the parentheses" pattern (`list.map { it * 2 }` instead of `list.map({ it -> it * 2 })`) shows up constantly in idiomatic Kotlin — not just for collections, but for things like building UI layouts in Jetpack Compose later on. Getting used to reading it now pays off throughout the rest of this course.

🎯 What's Next

Next chapter: **Null Safety** — nullable types, the safe call operator `?.`, the Elvis operator `?:`, smart casts, and the not-null assertion `!!`.

🛡️ Null Safety

Null safety is Kotlin's most famous feature, and the one most directly aimed at Java's biggest historical pain point — the NullPointerException. Kotlin bakes "can this be null?" directly into the type system, so the compiler catches most null-related bugs before the program ever runs. This chapter covers nullable types, the safe call operator, the Elvis operator, smart casts, and the escape hatch you should rarely reach for.

❓ Nullable Types

In Kotlin, a type is non-nullable by default. To allow `null`, the type must be explicitly marked with a trailing `?`:

```
var name: String = "Philip"
name = null // Compile error: String is non-nullable
var nickname: String? = "Phil"
nickname = null // Fine — String? explicitly allows null
```

This is the core idea of the whole chapter: `String` and `String?` are genuinely different types to the compiler. Every function, property, and variable declares up front whether `null` is a possibility — and the compiler then forces you to handle that possibility before you can use the value.

Null Handling: Java vs JavaScript vs Kotlin

Language	Can any reference be null?	Compiler enforcement
Java	Yes, any object reference	None — NPE is a runtime crash
JavaScript	Yes, plus a separate undefined	None — TypeError at runtime
Kotlin	Only types explicitly marked with ?	Compile error if unchecked

🔗 The Safe Call Operator: ?.

Calling a method or property directly on a nullable type is a compile error — Kotlin won't let you risk it. The safe call operator `?.` calls through only if the value isn't null, and evaluates to `null` otherwise:

```
val nickname: String? = null
// nickname.length        // Compile error: nickname might be null
println(nickname?.length)  // null — safely short-circuits, no crash
val realNickname: String? = "Phil"
println(realNickname?.length)  // 4
```

Safe calls chain naturally: `user?.address?.city` walks the chain and evaluates to `null` the moment any link in it is null, instead of crashing partway through — the same idea as JavaScript's optional chaining (`user?.address?.city`), which was directly inspired by this Kotlin feature.

🎸 The Elvis Operator: ?:

The Elvis operator provides a fallback value when the left-hand side is null — named for the way `?:` looks like Elvis Presley's hairstyle turned sideways:

```
val nickname: String? = null
val displayName = nickname ?: "Anonymous"
println(displayName)   // Anonymous
// Combined with safe call — a very common pattern:
val length = nickname?.length ?: 0
println(length)   // 0 — safe call gives null, Elvis supplies the fallback
```

?. — "call it if it's there"

Short-circuits to `null` instead of crashing when the receiver is null. Read `a?.b` as "get b from a, but only if a exists."

?: — "or fall back to this"

Supplies a default when the left side is null. Read `a ?: b` as "use a, or b if a is null." The two operators combine constantly: `a?.b ?: c`.

🎯 Smart Casts

After an explicit null check, the compiler "remembers" that a value can't be null within that scope, and lets you use it as the non-nullable type automatically — no manual cast required:

```
fun printLength(text: String?) {
    if (text != null) {
        // Inside this block, Kotlin smart-casts text from String? to String
println(text.length)   // no ?. needed here — compiler already knows it's not null
    } else {
        println("No text provided")
    }
}
```

This is one of Kotlin's quieter but most useful conveniences: Java requires an explicit cast even after a null check (or a local variable copy to avoid a "value could change" warning), while Kotlin's compiler tracks the null-checked state of a `val` automatically.

⚠ The Not-Null Assertion !! — Use Sparingly

The `!!` operator forces a nullable value to be treated as non-null, and throws a `NullPointerException` immediately if it turns out to be null after all: `nickname!!.length`. It exists for cases where you're certain a value can't be null but the compiler can't prove it — but reaching for it regularly defeats the entire point of Kotlin's null safety system. Prefer `?.` and `?:`; treat `!!` as a last resort, not a habit.

💻 Coding Challenges

Challenge 1: Safe Call Chains

Declare a nullable `val city: String? = null`. Use `println` with the safe call operator to print its `length` — it should print `null` without crashing. Then set `city` to `"London"` in a second variable and print its length again, this time getting an actual number.

Goal: See ?. short-circuit safely on both a null and a non-null value.

→ Solution

Challenge 2: Elvis Operator Defaults

Write a function `displayName(nickname: String?): String` that returns the nickname if present, or `"Anonymous"` otherwise, using the Elvis operator in a single-expression function. Call it with both a null and a non-null argument and print both results.

Goal: Practice ?: as a compact default-value pattern.

→ Solution

Challenge 3: Smart Casts

Write a function `describeAge(age: Int?)` that checks with an `if` whether `age` is not null. Inside the non-null branch, use the smart-cast value directly (no `?.` or `!!`) to print whether it's even or odd. In the else branch, print "Age unknown".

Goal: See the compiler treat a null-checked nullable value as non-nullable automatically.

→ Solution

💡 This Is Why Kotlin Won Over Android Developers

Null safety is consistently cited as the single biggest reason teams migrated from Java to Kotlin for Android work — NullPointerExceptions were historically one of the most common crash causes in Android apps. Moving null-checking from "something you remember to do" to "something the compiler enforces" eliminates an entire category of bugs at the source.

🎯 What's Next

Next chapter: **Classes & Data Classes** — constructors, properties, data classes, `copy()`, and companion objects.

Classes & Data Classes

Kotlin classes cut down dramatically on the boilerplate Java requires for even a simple class — no separate field declarations, no manually-written getters/setters, no repetitive constructor assignment. This chapter covers the primary constructor, properties, and the biggest boilerplate-killer of all: data classes, which generate an entire class's worth of standard methods from a single line.

🧱 The Primary Constructor

Kotlin classes declare their constructor parameters directly in the class header, and can turn them into properties with a single keyword:

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

val p = Person("Alice", 30)
println(p.name)   // Alice
println(p.age)    // 30
p.age = 31 // fine — age is a var property
// p.name = "Bob"     // compile error — name is a val property
```

`val` and `var` inside the constructor parentheses do two things at once: declare the parameter *and* promote it to a property of the class, automatically generating a getter (and a setter, for `var`). There's no separate field declaration, no `this.name = name` assignment, and no hand-written getter/setter — all of that Java ceremony collapses into the single constructor line.

A Simple Class: Java vs Kotlin

Java	Kotlin

public class Person {
  private final String name;
  private int age;
  public Person(String name, int age) {
  this.name = name;
  this.age = age;
  }
  public String getName() { return name; }
  public int getAge() { return age; }
  public void setAge(int age) { this.age = age; }
}
	class Person(val name: String, var age: Int)

⚙️ Init Blocks & Body Properties

Logic that needs to run at construction time goes in an `init` block; properties that aren't simple constructor parameters are declared in the class body:

```
class Person(val name: String, var age: Int) {
    val isAdult: Boolean
init {
        require(age >= 0) { "Age cannot be negative" }
        isAdult = age >= 18
println("Created Person: $name")
    }

    fun greet() = "Hi, I'm $name"
}
```

`require(...)` throws an `IllegalArgumentException` if the condition is false — Kotlin's built-in, one-line way of validating constructor arguments, instead of writing an explicit `if` and `throw`.

📋 Data Classes

Adding the `data` keyword to a class generates `equals()`, `hashCode()`, `toString()`, and `copy()` automatically, based on the constructor properties:

```
data class Point(val x: Int, val y: Int)

val a = Point(1, 2)
val b = Point(1, 2)

println(a)        // Point(x=1, y=2) — auto-generated toString()
println(a == b)   // true — auto-generated equals(), compares values not references
```

toString() — Free Debug Output

A plain Kotlin (or Java) class prints something unhelpful like `Point@6d06d69c` by default. A data class prints `Point(x=1, y=2)` — every property, readable, with zero code written.

equals() — Structural, Not Referential

A plain class's `==` compares references (are these the same object in memory?). A data class's `==` compares values — two separately-constructed `Point(1, 2)` instances are equal.

copy()

Data classes also generate a `copy()` function, which creates a new instance with the same values, except for whichever properties you explicitly override:

```
data class User(val name: String, val age: Int, val isAdmin: Boolean = false)

val original = User("Alice", 30)
val birthday = original.copy(age = 31)
val promoted = original.copy(isAdmin = true)

println(original)   // User(name=Alice, age=30, isAdmin=false) — unchanged
println(birthday)   // User(name=Alice, age=31, isAdmin=false)
println(promoted)   // User(name=Alice, age=30, isAdmin=true)
```

This pairs naturally with everything from Chapter 1 — data classes are usually built from `val` properties, so `copy()` is the standard way to produce an "updated" version without ever mutating the original. It's the same immutable-update pattern as JavaScript's spread syntax (`{ ...user, age: 31 }`), but type-checked and built into the class itself.

🏢 Companion Objects

Kotlin has no `static` keyword. Instead, members that belong to the class itself (not an instance) live inside a `companion object`:

```
class User private constructor(val name: String) {
    companion object {
        fun guest() = User("Guest")
    }
}

val g = User.guest()   // called on the class itself, like a Java static method
println(g.name)     // Guest
```

"Static" Members: Java vs Kotlin

Java	Kotlin
static User guest() { return new User("Guest"); }	companion object { fun guest() = User("Guest") }
User.guest()	User.guest()

The call syntax looks identical to Java's static call — `User.guest()` — but under the hood, a companion object is a real singleton object tied to the class, which is why it can also implement interfaces or hold its own properties, something a Java static block can't do.

💻 Coding Challenges

Challenge 1: A Class with an Init Block

Write a class `BankAccount(val owner: String, var balance: Double)` with an `init` block that uses `require` to reject a negative starting balance. Add a function `deposit(amount: Double)` that increases `balance`. Create an account, deposit twice, and print the final balance.

Goal: Combine constructor properties, init-block validation, and a regular method.

→ Solution

Challenge 2: Data Class Equality & toString

Declare a data class `Book(val title: String, val author: String, val year: Int)`. Create two separate `Book` instances with identical values and print whether they're equal with `==`. Print one of them directly and observe the auto-generated `toString()` output.

Goal: See structural equality and free toString() in action.

→ Solution

Challenge 3: copy() and a Companion Object

Using the same `Book` data class, use `copy()` to create a second-edition version of a book with an updated `year`, leaving the original unchanged. Then add a `companion object` with a function `unknown()` that returns a placeholder `Book("Unknown", "Unknown", 0)`, and call it via `Book.unknown()`.

Goal: Practice copy() for immutable updates and companion objects for factory-style functions.

→ Solution

💡 Data Classes Are Everywhere in Real Kotlin Code

Once Android chapters begin later on, data classes become the default shape for API response models, database entities, and UI state objects — anywhere a class exists mainly to hold values. The equals()/toString()/copy() trio this chapter covers isn't a niche feature; it's one of the most-used parts of the whole language.

🎯 What's Next

Next chapter: **Inheritance & Interfaces** — open/abstract classes, interfaces, polymorphism, and sealed classes.

🧬 Inheritance & Interfaces

Kotlin classes are closed to inheritance by default — the opposite of Java's default. This chapter covers how to deliberately open a class up for subclassing, abstract classes, interfaces, polymorphism, and sealed classes, which give Kotlin a safer alternative to open-ended inheritance hierarchies for a common category of problem.

🔒 Classes Are final by Default

```
class Animal(val name: String)

// class Dog : Animal("Rex")   // Compile error: Animal is final
```

In Java, every class can be subclassed unless it's explicitly marked `final`. Kotlin inverts that: every class is implicitly final unless explicitly marked `open`. This is a deliberate design choice — inheritance is powerful but easy to misuse, so Kotlin makes "can this be subclassed?" an explicit decision rather than an accident.

```
open class Animal(val name: String) {
    open fun makeSound() = "..."
}

class Dog(name: String) : Animal(name) {
    override fun makeSound() = "Woof!"
}

val pet: Animal = Dog("Rex")
println(pet.makeSound())   // Woof! — polymorphism: Animal reference, Dog behavior
```

Both the class (`open class Animal`) and the specific function being overridden (`open fun makeSound()`) need the `open` keyword — a class being open doesn't automatically make its members overridable. And the subclass must use `override` explicitly; Kotlin never lets an override happen silently.

Inheritance Defaults: Java vs Kotlin

	Java	Kotlin
Default	Open (subclassable)	Final (not subclassable)
Opt in to closing	final class Foo	class Foo (already final)
Opt in to opening	class Foo (already open)	open class Foo
Marking an override	@Override (annotation, optional)	override (keyword, required)

🧩 Abstract Classes

An `abstract` class can't be instantiated directly, and can declare members with no implementation, which subclasses are forced to provide:

```
abstract class Shape {
    abstract fun area(): Double
fun describe() = "This shape has an area of ${area()}"
}

class Circle(val radius: Double) : Shape() {
    override fun area() = Math.PI * radius * radius
}

// val s = Shape()        // Compile error: Shape is abstract, cannot be instantiated
println(Circle(2.0).describe())   // This shape has an area of 12.566...
```

Note that `abstract` members don't need the `open` keyword — being abstract already implies they must be overridden. `describe()` shows a common pattern: a concrete method in the abstract class that relies on the abstract method being implemented by whatever subclass eventually exists.

🔌 Interfaces

A Kotlin interface can declare abstract members (no body) and also provide default implementations, unlike a pre-Java-8 interface:

```
interface Drivable {
    val maxSpeed: Int // abstract property — implementer must provide it
fun drive()                 // abstract method
fun honk() = println("Beep!")  // default implementation — free for implementers
}

class Car(override val maxSpeed: Int) : Drivable {
    override fun drive() = println("Driving at up to $maxSpeed km/h")
}

val car = Car(180)
car.drive()   // Driving at up to 180 km/h
car.honk()    // Beep! — used as-is, never overridden
```

Multiple Interfaces, One Class

A class can implement any number of interfaces (`class Car : Drivable, Insurable`), unlike class inheritance, where Kotlin — like Java — allows only one superclass. Interfaces are how Kotlin achieves "multiple inheritance" of behavior.

Interface vs Abstract Class

Use an interface for a capability shared across unrelated classes (Drivable, Comparable). Use an abstract class when subclasses share actual state (constructor properties) and a genuine "is-a" relationship, not just a shared capability.

🔐 Sealed Classes

A `sealed` class restricts its subclasses to a known, fixed set, all declared in the same file. This is Kotlin's answer to representing "one of these specific things, and nothing else":

```
sealed class NetworkResult
data class Success(val data: String) : NetworkResult()
data class Error(val message: String) : NetworkResult()
object Loading : NetworkResult()

fun handle(result: NetworkResult) = when (result) {
    is Success -> println("Got: ${result.data}")
    is Error -> println("Failed: ${result.message}")
    Loading -> println("Loading...")
    // No "else" branch needed — the compiler knows these three are the only possibilities
}
```

⚠ The Real Payoff: Exhaustiveness Checking

Because a sealed class's subclasses are fixed and known, a `when` expression that covers every branch doesn't need (or allow, in expression form) an `else`. If a new subclass of `NetworkResult` is added later and this `when` isn't updated, the compiler flags it as an error — not a bug discovered at runtime. A plain interface or open class gives no such guarantee, since anyone, anywhere, could add a new implementation.

💻 Coding Challenges

Challenge 1: Open Class & Override

Write an `open class Employee(val name: String)` with an `open fun payDescription()` returning a generic string. Create a subclass `Manager(name: String, val bonus: Double) : Employee(name)` that overrides `payDescription()` to mention the bonus. Store a `Manager` in an `Employee`-typed variable and call `payDescription()` on it.

Goal: Practice open/override and see polymorphism through a superclass-typed reference.

→ Solution

Challenge 2: Interface with a Default Method

Write an interface `Greetable` with an abstract property `name: String`, an abstract function `customGreeting(): String`, and a default function `greet()` that prints `customGreeting()`. Implement it in two different classes with different greetings, and call `greet()` on each.

Goal: Combine abstract members with a default implementation in one interface.

→ Solution

Challenge 3: Sealed Class with when

Write a sealed class `Shape` with three subclasses: `data class Circle(val radius: Double)`, `data class Square(val side: Double)`, and `data class Rectangle(val width: Double, val height: Double)`. Write a function `area(shape: Shape): Double` using a `when` expression with no `else` branch that computes the correct area for each.

Goal: Practice sealed classes and see exhaustive when in action.

→ Solution

💡 Sealed Classes Preview Android's UI-State Pattern

The `NetworkResult` example above (Success / Error / Loading) is close to a verbatim preview of how UI state is modeled in real Android apps with Jetpack Compose — a screen's state is almost always a sealed class with exactly this shape. Getting comfortable with sealed classes and exhaustive when now sets up that pattern well ahead of time.

🎯 What's Next

Next chapter: **Collections & Functional Operators** — List/Set/Map, and the map/filter/reduce/flatMap operators used constantly in idiomatic Kotlin.

📚 Collections & Functional Operators

Kotlin's collections come in mutable and read-only flavors as a first-class distinction, and its functional operators — map, filter, reduce, flatMap — are the primary tool for working with them. If Chapter 2's lambdas felt abstract, this is where they become the everyday way real Kotlin code processes data.

📋 List, Set, and Map

```
val names: List<String> = listOf("Alice", "Bob", "Alice")
val uniqueNames: Set<String> = setOf("Alice", "Bob", "Alice")
val ages: Map<String, Int> = mapOf("Alice" to 30, "Bob" to 25)

println(names)       // [Alice, Bob, Alice] — duplicates kept, order preserved
println(uniqueNames) // [Alice, Bob] — duplicates removed
println(ages["Alice"]) // 30
```

`listOf`, `setOf`, and `mapOf` all produce **read-only** collections by default — no `add`, `remove`, or index-assignment is available on them. That's a deliberate mirror of Chapter 1's `val` vs `var` distinction, applied to collections: read-only is the default, mutability is opt-in.

Mutable Versions

`mutableListOf()`, `mutableSetOf()`, and `mutableMapOf()` return collections that support `add()`, `remove()`, and similar. Reach for these only when a collection genuinely needs to change after creation.

to — Building Pairs

`"Alice" to 30` creates a `Pair("Alice", 30)`. It's an infix function, not special syntax — `mapOf` is really just a function that accepts a list of `Pair`s.

Collection Literals: JavaScript vs Kotlin

	JavaScript	Kotlin
List/Array	const names = ["Alice", "Bob"];	val names = listOf("Alice", "Bob")
Map/Object	const ages = {Alice: 30, Bob: 25};	val ages = mapOf("Alice" to 30, "Bob" to 25)
Read-only by default?	No — arrays/objects are always mutable	Yes — listOf/mapOf are read-only

🗺️ map — Transform Every Element

```
val numbers = listOf(1, 2, 3, 4)
val squared = numbers.map { it * it }
println(squared)  // [1, 4, 9, 16]
```

`map` applies the lambda to every element and returns a new list of the results, one-to-one with the input — the same shape as JavaScript's `Array.prototype.map`.

🔍 filter — Keep Some Elements

```
val numbers = listOf(1, 2, 3, 4, 5, 6)
val evens = numbers.filter { it % 2 == 0 }
println(evens)   // [2, 4, 6]
```

`filter` keeps only elements where the lambda returns `true` — Kotlin's equivalent of JavaScript's `Array.prototype.filter`, with the same predicate-based idea.

➕ reduce — Combine into One Value

```
val numbers = listOf(1, 2, 3, 4)

val sum = numbers.reduce { acc, n -> acc + n }
println(sum)   // 10
// fold is like reduce, but takes an explicit starting value
val sumFrom100 = numbers.fold(100) { acc, n -> acc + n }
println(sumFrom100)   // 110
```

`reduce` walks the list, accumulating a single result — `acc` starts as the first element, then each step combines it with the next. `fold` is nearly identical but takes an explicit initial value instead of using the first element, which also means `fold` works safely on an empty list (`reduce` throws on an empty collection, since it has no first element to start from).

🌊 flatMap — Map, Then Flatten

```
val teams = listOf(listOf("Alice", "Bob"), listOf("Carol", "Dave"))

val allPlayers = teams.flatMap { it }
println(allPlayers)  // [Alice, Bob, Carol, Dave] — one flat list
// Compare to plain map, which keeps the nesting:
val stillNested = teams.map { it }
println(stillNested)  // [[Alice, Bob], [Carol, Dave]]
```

`flatMap` is `map` followed by flattening one level of nesting — useful whenever the transformation for each element produces its own list, and the goal is a single combined list rather than a list of lists.

Functional Operators Quick Reference

Operator	Input → Output	JavaScript Equivalent
map	N elements → N transformed elements	Array.prototype.map
filter	N elements → subset matching a condition	Array.prototype.filter
reduce / fold	N elements → 1 accumulated value	Array.prototype.reduce
flatMap	N elements → N lists, flattened into 1	Array.prototype.flatMap

⚡ Sequences — Lazy Evaluation

Chaining several operators on a regular collection (`List`, `Set`) creates a full intermediate list at *every* step. A `Sequence` instead evaluates lazily, element by element, through the whole chain at once:

```
val result = (1..1_000_000)
    .asSequence()
    .filter { it % 2 == 0 }
    .map { it * it }
    .take(5)
    .toList()

println(result)  // [4, 16, 36, 64, 100] — only computed enough elements to satisfy take(5)
```

⚠ Without asSequence(), Every Step Builds a Full List

The equivalent chain without `.asSequence()` — `(1..1_000_000).filter { ... }.map { ... }.take(5)` — would first filter *all* one million numbers into a 500,000-element list, then map *all* of those, and only then take 5. On large or infinite ranges, that gap in work done is significant; on small lists (the vast majority of real code), plain operators are simpler and the difference doesn't matter.

💻 Coding Challenges

Challenge 1: List, Set, and Map Basics

Create a `List<String>` of five city names including one duplicate, convert it to a `Set` to remove the duplicate, and print both. Then create a `Map<String, Int>` of three cities to their population (in millions) and print the population of one city by key lookup.

Goal: Get comfortable creating and reading from all three core collection types.

→ Solution

Challenge 2: filter, map, and reduce Chained

Given `val prices = listOf(12.50, 8.00, 25.00, 3.50, 40.00)`, filter to prices over 10.0, map each remaining price by applying a 20% discount, then use `reduce` to sum the discounted prices into a single total. Print the total.

Goal: Chain three functional operators together in one realistic calculation.

→ Solution

Challenge 3: flatMap

Given `val orders = listOf(listOf("Apple", "Bread"), listOf("Milk"), listOf("Eggs", "Cheese", "Butter"))` (each inner list is one customer's order), use `flatMap` to produce a single flat list of every item across all orders, then print how many total items were ordered using `.size`.

Goal: Practice flattening a list of lists into one combined list.

→ Solution

💡 Chaining Reads Like a Pipeline

Kotlin code very rarely writes a manual for-loop to transform data — chains like `list.filter { ... }.map { ... }.sortedBy { ... }` are the idiomatic default. Each operator returns a new collection, so the whole chain reads top-to-bottom as a data pipeline: start with the source, keep applying steps, end with the result.

🎯 What's Next

Next chapter: **Control Flow & Destructuring** — `when` expressions, ranges, loops, and destructuring declarations.

🔀 Control Flow & Destructuring

This chapter rounds out the fundamentals with Kotlin's expression-oriented control flow — where if and when produce values rather than just branching — plus ranges, loop syntax, and destructuring declarations, which unpack a value into several variables in one line.

🎛️ when — Kotlin's Souped-Up switch

```
val day = 3
val dayName = when (day) {
  1 -> "Monday"
2 -> "Tuesday"
3 -> "Wednesday"
6, 7 -> "Weekend"
else -> "Unknown day"
}
println(dayName)  // Wednesday
```

`when` is a direct upgrade over Java/JavaScript's `switch`: no `break` is needed (there's no fallthrough to guard against), a single branch can match multiple values (`6, 7 ->`), and — critically — `when` is an **expression**, meaning it produces a value that can be assigned directly, as shown above.

```
// when without an argument — replaces a long if/else-if chain
val score = 85
val grade = when {
  score >= 90 -> "A"
  score >= 80 -> "B"
  score >= 70 -> "C"
else -> "F"
}
println(grade)  // B
```

Without a subject in parentheses, each branch is a boolean condition evaluated top to bottom — the first one that's `true` wins. This form directly replaces a chain of `if / else if / else if / else`, and reads more clearly once there are more than two or three conditions.

switch vs when

	Java/JavaScript switch	Kotlin when
Fallthrough	Falls through without break	Never falls through
Multiple values, one branch	case 6: case 7: …	6, 7 -> …
Produces a value?	No — statement only	Yes — can be an expression
Condition-only form	Not available (need if/else-if)	when { cond -> … }

🔢 Ranges

```
for (i in 1..5) print("$i ")        // 1 2 3 4 5
println()

for (i in 1 until 5) print("$i ")    // 1 2 3 4 — excludes 5
println()

for (i in 5 downTo 1) print("$i ")   // 5 4 3 2 1
println()

for (i in 1..10 step 2) print("$i ") // 1 3 5 7 9
```

`1..5` creates an inclusive range (both ends included). `until` makes the upper bound exclusive — the direct equivalent of a classic `for (int i = 0; i < 5; i++)` loop. `downTo` counts backward, and `step` changes the increment. Ranges aren't just for loops either — `if (age in 13..19) println("Teenager")` reads naturally as a membership check.

🔁 Loops

```
val fruits = listOf("apple", "banana", "cherry")

// for-each — the most common loop by far
for (fruit in fruits) {
    println(fruit)
}

// index + value together, via withIndex()
for ((index, fruit) in fruits.withIndex()) {
    println("$index: $fruit")
}

// while — identical to Java/JavaScript
var countdown = 3
while (countdown > 0) {
    println(countdown)
    countdown--
}
```

The `for (item in collection)` form is Kotlin's only `for` syntax — there's no classic C-style `for (int i = 0; i < n; i++)` loop at all. Combined with ranges, the same job is done with `for (i in 0 until n)`, and combined with `withIndex()`, index-and-value iteration reads naturally without manual index bookkeeping.

📦 Destructuring Declarations

A value can be "unpacked" into several variables in one line, if the type supports it — data classes support this automatically, based on constructor property order:

```
data class Point(val x: Int, val y: Int)

val point = Point(3, 7)
val (x, y) = point

println("x=$x, y=$y")  // x=3, y=7
```

The earlier `withIndex()` loop was already destructuring in action: `(index, fruit)` unpacks each pair the loop produces. `Map` iteration destructures the same way:

```
val ages = mapOf("Alice" to 30, "Bob" to 25)

for ((name, age) in ages) {
    println("$name is $age")
}
// Alice is 30
// Bob is 25
```

Ignoring a Component: _

An underscore skips a component you don't need: `val (x, _) = point` unpacks only `x`, discarding `y` without needing to name an unused variable.

Order Matters, Not Names

Destructuring is positional — it maps to a data class's constructor properties in declared order, regardless of what you name the variables on the left-hand side.

⚠ Only Works on Destructuring-Capable Types

Destructuring relies on the type providing `component1()`, `component2()`, etc. — data classes generate these automatically, and `Pair`/`Map.Entry` provide them too, which is why `for ((k, v) in map)` works. A plain (non-data) class does not support this syntax unless those component functions are written manually.

💻 Coding Challenges

Challenge 1: when as an Expression

Write a function `seasonFor(month: Int): String` that uses a `when` expression (with grouped values, e.g. `12, 1, 2 -> "Winter"`) to return the season name for a given month number (1-12), directly returning the when result with no explicit return statement.

Goal: Use when as an expression, including grouped-value branches.

→ Solution

Challenge 2: Ranges and Loops

Using a range-based for loop, print every odd number from 1 to 20 using `step`. Then, given `val colors = listOf("red", "green", "blue")`, use `withIndex()` to print each color with its index in the format `0: red`.

Goal: Combine range step with index-and-value loop iteration.

→ Solution

Challenge 3: Destructuring a Map

Create a `data class Product(val name: String, val price: Double)`, destructure an instance into `name` and `price` variables and print them. Then create a `Map<String, Double>` of three product names to prices, and use a destructuring for-loop to print each as "name: $price".

Goal: Practice destructuring both a data class instance and a Map in a loop.

→ Solution

💡 Course 1 Is Nearly Complete

This chapter closes out the core language fundamentals — types, functions, null safety, classes, collections, and control flow. The final chapter of this course covers extension functions and object declarations, two features that round out how idiomatic Kotlin code is actually structured day to day.

🎯 What's Next

Next chapter: **Extension Functions & Object Declarations** — extending existing classes, singletons, and object expressions.

🧰 Extension Functions & Object Declarations

The final chapter of Kotlin Fundamentals covers two features that shape how idiomatic Kotlin code is actually organized: extension functions, which add new behavior to existing classes without touching or subclassing them, and object declarations, Kotlin's built-in, thread-safe way of writing a singleton.

➕ Extension Functions

An extension function adds a new method to an existing class — including classes you don't own, like `String` or `Int` — without modifying its source or subclassing it:

```
fun String.shout(): String = uppercase() + "!"
println("hello".shout())  // HELLO!
```

The syntax is `fun ReceiverType.functionName()` — the type before the dot is called the **receiver type**. Inside the function, `this` refers to the specific string it was called on, exactly as if `shout()` had been a real member of `String` all along. Calling it (`"hello".shout()`) looks completely indistinguishable from calling a built-in method.

Why Not Just Subclass?

`String` is a final, built-in class — it can't be subclassed at all. Even for classes you do own, extension functions let you add a helper without bloating the original class definition or creating an inheritance relationship that doesn't really exist.

Resolved at Compile Time

Unlike a real override, which method actually runs is decided statically, based on the declared type — extension functions aren't part of the class itself, and don't participate in polymorphism the way a real method does.

```
// A practical example: an extension on your own data class
data class Product(val name: String, val price: Double)

fun Product.priceWithTax(rate: Double = 0.2): Double = price * (1 + rate)

val item = Product("Mouse", 20.0)
println(item.priceWithTax())  // 24.0
```

Adding Behavior to an Existing Type: Java vs Kotlin

Java	Kotlin
static String shout(String s) { return s.toUpperCase() + "!"; } shout("hello")	fun String.shout() = uppercase() + "!" "hello".shout()

Java's usual workaround is a static utility method (`StringUtils.shout(s)`) — functionally similar, but called with the value as an argument rather than as the receiver, so it doesn't read as naturally at the call site and doesn't show up in IDE autocomplete when typing `someString.`.

🏢 object — Kotlin's Built-In Singleton

Chapter 4 introduced `companion object` for per-class "static" members. A plain `object` declaration is the more general form — an entire class with exactly one instance, guaranteed:

```
object AppConfig {
    val appName = "Learning Blog"
var debugMode = false
fun printInfo() = println("$appName (debug=$debugMode)")
}

AppConfig.debugMode = true
AppConfig.printInfo()   // Learning Blog (debug=true)
```

There's no constructor, and no way to create a second instance — Kotlin creates the single instance lazily on first access and guarantees it's thread-safe, with no extra code required. This directly replaces the classic Java singleton pattern (a private constructor, a static instance field, a static `getInstance()` method, and manual thread-safety handling) with one keyword.

⚠ object vs class — Choosing Correctly

Use `object` only when there's genuinely one, shared instance for the entire app's lifetime (configuration, a shared cache, a stateless utility holder). If you find yourself wanting more than one instance later, or passing constructor arguments in, that's a sign it should have been a regular `class` all along.

🎭 Object Expressions

An anonymous, one-off object can also be created inline, without a named declaration — most often to implement an interface on the spot:

```
interface ClickListener {
    fun onClick()
}

fun setListener(listener: ClickListener) {
    listener.onClick()
}

setListener(object : ClickListener {
    override fun onClick() {
        println("Clicked!")
    }
})
```

This is Kotlin's equivalent of Java's anonymous inner class (`new ClickListener() { ... }`) — a throwaway implementation created exactly where it's used, with no separate named class needed. It shows up constantly in UI callback code, including Android's older View-based click listeners.

💻 Coding Challenges

Challenge 1: Extension Function on Int

Write an extension function `fun Int.isPrime(): Boolean` that returns whether the receiver is a prime number (handle 2 and up; anything less than 2 is not prime). Test it against several numbers, calling it as `7.isPrime()`.

Goal: Add a genuinely new capability to a built-in type via an extension function.

→ Solution

Challenge 2: A Singleton with object

Write an `object Counter` with a `var count: Int = 0` property and a function `increment()` that adds 1. Call `increment()` three times from `main()` and print the final count, then explain in a comment why this couldn't be done the same way with a regular class without extra setup.

Goal: Use object for genuine shared, single-instance state.

→ Solution

Challenge 3: Object Expression Implementing an Interface

Write an interface `Validator` with a function `isValid(input: String): Boolean`. Write a function `checkInput(text: String, validator: Validator)` that prints whether the input is valid. Call it by passing an inline object expression that checks the input isn't blank and is at least 3 characters long.

Goal: Implement an interface with an anonymous object expression instead of a named class.

→ Solution

💡 Course 1 Complete — What This Sets Up

That's every chapter of Kotlin Fundamentals — types, functions, null safety, classes, inheritance, collections, control flow, and now extension functions and singletons. Kotlin Intermediate (Course 2) builds directly on all of this, starting with coroutines — Kotlin's approach to asynchronous code, and one of its most distinctive features next to null safety itself.

🎯 What's Next

Course 1 is complete. Kotlin Intermediate (Course 2) begins with **Coroutines Basics** — suspend functions, CoroutineScope, launch/async, and Dispatchers.