**Section 46 — Kotlin Basics**

**1. Introduction to Kotlin**

* **Modern, concise language** officially supported by Google for Android.
* Fully **interoperable with Java** → can use Java libraries/classes.
* Eliminates common Java issues (null pointer exceptions, boilerplate code).
* **Primary use in Android**: Shorter, safer, and cleaner code.

**2. Setting up Kotlin in Android Studio**

* Kotlin is **already built-in** in the latest Android Studio.
* To add Kotlin support in an existing project:
  1. Tools → Kotlin → Configure Kotlin in Project
  2. Gradle will add:

gradle

CopyEdit

plugins {

id 'org.jetbrains.kotlin.android'

}

* **Best practice:** Start new Android projects with "Language = Kotlin" option.

**3. Variables & Data Types**

* **Immutable (val)** → Value cannot be changed (similar to final in Java).

kotlin

CopyEdit

val name = "Harsh" // cannot reassign

* **Mutable (var)** → Value can be changed.

kotlin

CopyEdit

var age = 22

age = 23

* **Type Inference** → Kotlin can guess variable type automatically.

kotlin

CopyEdit

val score = 90 // Int

* **Explicit Type** (optional):

kotlin

CopyEdit

var salary: Double = 50000.0

**4. Null Safety**

* Kotlin helps avoid NullPointerException (NPE) by:
  + Variables are **non-null by default**.
  + To allow null values, use ? after type.

kotlin

CopyEdit

var address: String? = null

* Safe call operator (?.) → Executes only if variable is non-null.

kotlin

CopyEdit

println(address?.length)

* Elvis operator (?:) → Provide default if null.

kotlin

CopyEdit

val length = address?.length ?: 0

* Non-null assertion (!!) → Force treat as non-null (can crash).

kotlin

CopyEdit

println(address!!.length)

**5. Functions**

* Basic function:

kotlin

CopyEdit

fun greet(name: String): String {

return "Hello, $name"

}

* Single-expression function:

kotlin

CopyEdit

fun add(a: Int, b: Int) = a + b

* **Default arguments**:

kotlin

CopyEdit

fun welcome(name: String = "Guest") {

println("Welcome $name")

}

* **Named arguments**:

kotlin

CopyEdit

welcome(name = "Harsh")

**6. String Templates**

* Directly insert variables or expressions:

kotlin

CopyEdit

val user = "Harsh"

println("Hello $user")

println("Next year: ${age + 1}")

**7. Conditional Statements**

* If-else:

kotlin

CopyEdit

if (score > 50) {

println("Pass")

} else {

println("Fail")

}

* If as expression:

kotlin

CopyEdit

val result = if (score > 50) "Pass" else "Fail"

* when statement (like switch in Java):

kotlin

CopyEdit

when (day) {

1 -> println("Monday")

2 -> println("Tuesday")

else -> println("Other day")

}

**8. Loops**

* For loop:

kotlin

CopyEdit

for (i in 1..5) { // inclusive

println(i)

}

for (i in 1 until 5) { // exclusive of 5

println(i)

}

* While loop:

kotlin

CopyEdit

var x = 5

while (x > 0) {

println(x)

x--

}

**9. Collections**

* **List** (immutable by default):

kotlin

CopyEdit

val names = listOf("A", "B", "C")

* **MutableList**:

kotlin

CopyEdit

val fruits = mutableListOf("Apple", "Banana")

fruits.add("Orange")

* **Set** → No duplicates.

kotlin

CopyEdit

val ids = setOf(1, 2, 2, 3) // result: 1, 2, 3

* **Map** → Key-value pairs.

kotlin

CopyEdit

val countryCodes = mapOf("IN" to "India", "US" to "United States")

**10. Classes & Objects**

* Class:

kotlin

CopyEdit

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

* Object creation:

kotlin

CopyEdit

val p1 = Person("Harsh", 22)

* **Data classes** → Auto-generate toString(), equals(), etc.

kotlin

CopyEdit

data class User(val id: Int, val name: String)

**11. Inheritance**

* Classes are final by default → use open to allow inheritance.

kotlin

CopyEdit

open class Animal {

open fun sound() {

println("Animal sound")

}

}

class Dog: Animal() {

override fun sound() {

println("Bark")

}

}

**12. Interfaces**

* Define methods without implementation.

kotlin

CopyEdit

interface Clickable {

fun onClick()

}

class Button: Clickable {

override fun onClick() {

println("Button clicked")

}

}

**13. Companion Objects & Static Members**

* Kotlin has no static keyword.
* Use companion object to hold members tied to the class, not instances.

kotlin

CopyEdit

class Utils {

companion object {

const val PI = 3.14

fun printMessage() = println("Hello")

}

}

Utils.printMessage()

**14. Extension Functions**

* Add new functions to existing classes without inheritance.

kotlin

CopyEdit

fun String.addExclamation() = this + "!"

println("Hello".addExclamation())

**15. Lambda Functions**

* Anonymous functions for short operations.

kotlin

CopyEdit

val sum = { a: Int, b: Int -> a + b }

println(sum(5, 3))

**Tools, Libraries, APIs**

* **Kotlin Standard Library** (no extra install needed).
* **Android Studio built-in Kotlin support**.
* **Kotlinx Coroutines** (for async programming, not covered here but essential later).

**Best Practices**

* Prefer val over var for immutability.
* Avoid !! unless absolutely sure variable is not null.
* Use string templates instead of concatenation.
* Use when for cleaner branching.
* Use data classes for model objects.
* Use apply, let, run, also for cleaner object configuration.

**Part B — Missing but Important**

**1. Kotlin Coroutines**

* **What:** A modern way to handle asynchronous tasks (network calls, database queries) without blocking the main thread.
* **Why important:** In Android, UI must remain responsive — coroutines simplify async code compared to callbacks or RxJava.
* **How to implement:**

kotlin

CopyEdit

import kotlinx.coroutines.\*

fun main() {

GlobalScope.launch { // launches a coroutine

delay(1000L) // non-blocking delay

println("Hello from coroutine!")

}

println("Main thread continues...")

Thread.sleep(2000L) // wait for coroutine to finish

}

* **Best practice in Android:** Use viewModelScope for coroutines in ViewModel, lifecycleScope for UI components.
* **Example in Android:**

kotlin

CopyEdit

lifecycleScope.launch {

val data = withContext(Dispatchers.IO) { fetchFromNetwork() }

updateUI(data)

}

**2. Sealed Classes**

* **What:** Special classes for representing **restricted type hierarchies** — all subclasses are known at compile-time.
* **Why important:** Useful for handling multiple UI states in Android (e.g., Loading, Success, Error) with when.
* **Example:**

kotlin

CopyEdit

sealed class Result {

data class Success(val data: String) : Result()

data class Error(val message: String) : Result()

object Loading : Result()

}

fun handle(result: Result) {

when (result) {

is Result.Success -> println("Data: ${result.data}")

is Result.Error -> println("Error: ${result.message}")

Result.Loading -> println("Loading...")

}

}

**3. Higher-Order Functions**

* **What:** Functions that take other functions as parameters or return a function.
* **Why important:** Simplifies repetitive code patterns (e.g., click listeners, list filtering).
* **Example:**

kotlin

CopyEdit

fun operate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {

return operation(a, b)

}

fun main() {

val sum = operate(3, 4) { x, y -> x + y }

println(sum) // Output: 7

}

**4. Scope Functions (let, apply, also, with, run)**

* **What:** Built-in Kotlin functions for cleaner object manipulation.
* **Why important:** Reduces boilerplate when initializing, modifying, or using objects.
* **Summary Table:**

| **Function** | **Context** | **Return** | **Use case** |
| --- | --- | --- | --- |
| let | it | Lambda result | Null-check, chain calls |
| apply | this | Object itself | Object configuration |
| also | it | Object itself | Extra side effects |
| with | this | Lambda result | Group operations on object |
| run | this | Lambda result | Combination of let and with |

* **Example:**

kotlin

CopyEdit

data class User(var name: String, var age: Int)

val user = User("Harsh", 22).apply {

age += 1 // modify directly

}.also {

println("User updated: $it")

}

**5. Delegated Properties**

* **What:** Property value handling delegated to another object.
* **Why important:** Useful for lazy initialization, observable changes, or storing in custom locations.
* **Example — Lazy initialization:**

kotlin

CopyEdit

val heavyValue: String by lazy {

println("Computed!")

"Hello World"

}

fun main() {

println(heavyValue) // First access → computes value

println(heavyValue) // Next access → uses cached value

}

* **Example — Observable:**

kotlin

CopyEdit

import kotlin.properties.Delegates

var name: String by Delegates.observable("<no name>") { prop, old, new ->

println("$old -> $new")

}

**6. Destructuring Declarations**

* **What:** Extract multiple properties from an object into separate variables.
* **Why important:** Clean handling of data classes and collections.
* **Example:**

kotlin

CopyEdit

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

val person = Person("Harsh", 22)

val (n, a) = person

println("$n is $a years old")

**7. Kotlin Flow**

* **What:** A coroutine-based reactive stream — emits multiple values over time.
* **Why important:** Modern replacement for LiveData in many cases.
* **Example:**

kotlin

CopyEdit

import kotlinx.coroutines.\*

import kotlinx.coroutines.flow.\*

fun numbersFlow(): Flow<Int> = flow {

for (i in 1..3) {

delay(1000)

emit(i)

}

}

fun main() = runBlocking {

numbersFlow().collect { value ->

println("Received $value")

}

}

**8. Inline Classes**

* **What:** Lightweight wrappers around a value type without additional memory overhead.
* **Why important:** Type safety without runtime penalty.
* **Example:**

kotlin

CopyEdit

@JvmInline

value class UserId(val id: String)

fun printUser(id: UserId) {

println("User ID: ${id.id}")

}