**Section 49: Lambda Expressions, Anonymous & Higher Order Functions**

**0. Key Concepts Taught**

1. **Literals**
   * Direct representations of constant values in source code.
   * Examples: Integer (10), Float (3.14), String ("Hello").
   * Used for initializing variables, constants, properties.
2. **Function Literals** (Lambda Expressions & Anonymous Functions)
   * Functions without a name.
   * Can be passed as parameters to other functions.
   * Promote **functional programming style** in Kotlin.
   * Used with **higher order functions** like map, filter, reduce, forEach.
3. **Lambda Expressions**
   * Concise syntax for inline functions.
   * Can be stored in variables, passed to functions, or executed directly.
   * Syntax:
   * val name: (ParamType1, ParamType2) -> ReturnType = { param1, param2 -> body }
   * Types:
     1. With parameters + return value.
     2. With parameters + no return value.
     3. No parameters + return value.
     4. No parameters + no return value.
   * Can be called using () or .invoke().
4. **Anonymous Functions**
   * Similar to lambdas but explicitly use fun keyword.
   * Can explicitly define return type.
   * Syntax:
   * val funcName = fun(param: Type): ReturnType {
   * return ...
   * }
   * Four types (same as lambdas, based on parameters/return type).
5. **Higher Order Functions**
   * Functions that take functions as parameters or return functions.
   * Enable **behavior injection** and reusability.
   * Example:
   * fun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
   * return op(a, b)
   * }
6. **it Keyword**
   * Implicit name for a single parameter in a lambda/anonymous function.
   * Can be used when there’s only **one parameter**.
   * Makes code shorter:
   * numbers.map { it \* it }

**1. Core Concepts**

| **Concept** | **What It Means** | **Key Benefit** |
| --- | --- | --- |
| **Literals** | Direct representations of constant values in source code (e.g., 10, 3.14, "Hello"). | Fast initialization of variables and constants. |
| **Function Literals** | Functions without names (lambdas & anonymous functions). | Pass functions as data for flexibility. |
| **Lambda Expression** | Concise function literal written in {}. | Short inline code without fun keyword. |
| **Anonymous Function** | Function literal using fun. | Can specify return type & use multiple return points. |
| **Higher Order Function (HOF)** | Function that takes or returns another function. | Enables reusable logic and functional patterns. |
| **it keyword** | Implicit single-parameter name in lambdas. | Reduces boilerplate when only one parameter is used. |

**2. Syntax Patterns with All Combinations**

**Lambda Expressions**

// General Syntax

val name: (ParamType1, ParamType2) -> ReturnType = { param1, param2 ->

// body

}

// 1️⃣ Parameters + Return value

val add: (Int, Int) -> Int = { a, b -> a + b }

// 2️⃣ Parameters + No return (Unit)

val printSum: (Int, Int) -> Unit = { a, b -> println("Sum is ${a + b}") }

// 3️⃣ No parameters + Return value

val greet: () -> String = { "Welcome to Kotlin" }

// 4️⃣ No parameters + No return

val sayHello: () -> Unit = { println("Hello!") }

// Calling

println(add(5, 3))

println(add.invoke(5, 3))

**Anonymous Functions**

// General Syntax

val func = fun(param1: Type1, param2: Type2): ReturnType {

// body

}

// Examples

val multiply = fun(a: Int, b: Int): Int { return a \* b }

val printProduct = fun(a: Int, b: Int) { println("Product: ${a \* b}") }

val welcome = fun(): String { return "Hello from anonymous function!" }

val noOp = fun() { println("Doing nothing") }

**Higher Order Functions**

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

return operation(a, b)

}

val sumResult = operateOnNumbers(5, 3) { x, y -> x + y }

val productResult = operateOnNumbers(5, 3) { x, y -> x \* y }

println("Sum: $sumResult")

println("Product: $productResult")

**it Keyword**

val numbers = listOf(1, 2, 3, 4, 5)

val squared = numbers.map { it \* it }

val evens = numbers.filter { it % 2 == 0 }

**3. Real-World Android Use Cases**

**RecyclerView Adapter Click Listener**

class MyAdapter(

private val items: List<String>,

private val onClick: (String) -> Unit

) : RecyclerView.Adapter<MyViewHolder>() {

override fun onBindViewHolder(holder: MyViewHolder, position: Int) {

holder.itemView.setOnClickListener { onClick(items[position]) }

}

}

// Usage

adapter = MyAdapter(data) { item ->

Toast.makeText(context, "Clicked $item", Toast.LENGTH\_SHORT).show()

}

**Button Click**

myButton.setOnClickListener {

Toast.makeText(this, "Button clicked", Toast.LENGTH\_SHORT).show()

}

**Firebase Callback**

db.collection("users")

.get()

.addOnSuccessListener { result ->

result.forEach { doc -> println(doc.data) }

}

**4. Performance Tips**

* **Use inline for HOFs** to avoid object creation:

inline fun measureTime(block: () -> Unit) {

val start = System.currentTimeMillis()

block()

println("Time: ${System.currentTimeMillis() - start}ms")

}

* Avoid capturing large objects (e.g., Activity) in lambdas → may cause memory leaks.
* Prefer function references (::myFunction) over lambdas when possible.

**5. Best Practices (Expanded)**

1. **Prefer lambdas for short logic** (1–3 lines) to keep code concise.
2. **Use anonymous functions** when:
   * Explicit return type needed.
   * Multiple return points in function body.
3. **Use it only for single parameters**; otherwise, name them clearly.
4. **Combine lambdas with Kotlin’s collection API** (map, filter, reduce, groupBy).
5. **Inline HOFs** in performance-sensitive code (network parsing, loops).
6. **Avoid storing lambdas holding Activity or Fragment** to prevent leaks.

**Part B — Extra Knowledge for Future Projects**

**Inline Function Modifiers**

* noinline: Keeps some lambdas as callable objects.
* crossinline: Prevents non-local return from lambdas.

**SAM Conversions**

* Pass lambdas to Java single-method interfaces.

Thread { println("In thread") }.start()

**Function References**

fun isEven(n: Int) = n % 2 == 0

val evens = listOf(1, 2, 3, 4).filter(::isEven)

**Closures**

fun makeMultiplier(factor: Int): (Int) -> Int {

return { it \* factor }

}

**Chaining**

val result = (1..10)

.map { it \* it }

.filter { it > 20 }

.sum()

**Currying & Partial Application**

fun add(a: Int) = { b: Int -> a + b }

val addFive = add(5)

**Coroutines**

GlobalScope.launch {

delay(1000)

println("Done")

}

✅ With these notes, you can:

* Recognize every form of lambda & anonymous function.
* Use HOFs effectively in Android projects.
* Apply functional patterns for cleaner Kotlin code.
* Avoid performance pitfalls & memory leaks.