**Section 75 — Coroutines in Kotlin: Master Guide**

**1. Key Concepts Taught**

| **Concept** | **Explanation** | **Why It Matters** |
| --- | --- | --- |
| **Coroutine** | A lightweight thread for asynchronous programming in Kotlin. | Allows writing async, non-blocking code that looks sequential. |
| **Suspend Function** | A function marked with suspend keyword that can be paused and resumed. | Used for executing long-running tasks without blocking the thread. |
| **CoroutineScope** | Defines the lifecycle and context for coroutines. | Prevents leaks; coroutines cancel automatically when scope ends. |
| **Dispatchers** | Decide the thread on which coroutine runs (Main, IO, Default). | Optimizes performance by using correct threads for tasks. |
| **launch vs async** | launch → fire-and-forget, returns Job; async → returns Deferred for results. | Choose based on whether you need a result. |
| **runBlocking** | Starts a coroutine in a blocking way (for main functions or testing). | Only for special cases, not in production UI code. |

**2. Step-by-Step Implementation**

**A. Basic Coroutine Launch**

import kotlinx.coroutines.\*

// Entry point

fun main() = runBlocking {

// launch starts a new coroutine without blocking the current thread

launch {

delay(1000L) // Simulates network/API delay

println("Task 1 done on ${Thread.currentThread().name}")

}

println("Main program continues...") // Runs immediately

}

**Explanation:**

* runBlocking creates a coroutine scope and blocks the main thread until all coroutines finish.
* launch starts a coroutine in parallel.
* delay() is a **non-blocking** suspend function (releases the thread during wait).

**B. Using Dispatchers**

fun main() = runBlocking {

launch(Dispatchers.Main) {

// For UI work (Android main thread)

println("Running on Main thread")

}

launch(Dispatchers.IO) {

// For disk/network I/O

println("Running on IO thread")

}

launch(Dispatchers.Default) {

// For CPU-intensive work

println("Running on Default thread")

}

}

**Why important in Android:**

* Prevents **ANR** (App Not Responding) by moving heavy work off the main thread.
* Each dispatcher uses optimized thread pools.

**C. Suspend Functions**

suspend fun fetchData(): String {

delay(2000L) // Simulates network call

return "Data fetched!"

}

fun main() = runBlocking {

val data = fetchData()

println(data)

}

**Notes:**

* Can only call suspend functions from another suspend function or coroutine.
* delay here is **non-blocking**, unlike Thread.sleep().

**D. async for Parallel Tasks**

fun main() = runBlocking {

val task1 = async {

delay(1000L)

"Result from Task 1"

}

val task2 = async {

delay(2000L)

"Result from Task 2"

}

// Await results in parallel

println("${task1.await()} & ${task2.await()}")

}

**Why:**

* async starts tasks immediately and returns a Deferred.
* await() gets the result **without blocking**.

**E. Cancelling Coroutines**

fun main() = runBlocking {

val job = launch {

repeat(5) { i ->

println("Working $i ...")

delay(500L)

}

}

delay(1200L)

job.cancel() // Cancels coroutine

println("Job cancelled")

}

**Why:**

* Essential for Android lifecycle (cancel network calls when Activity is destroyed).

**3. Tools, Libraries, and APIs**

* **Kotlinx Coroutines Library** (org.jetbrains.kotlinx:kotlinx-coroutines-core)
* **Dispatchers** for threading:
  + Main → UI updates
  + IO → disk/network
  + Default → CPU-heavy work
* **LifecycleScope / ViewModelScope** in AndroidX for lifecycle-aware coroutines.

**4. Best Practices (Expanded)**

1. **Always Use Lifecycle-Aware Scopes in Android**
   * lifecycleScope for Activities/Fragments.
   * viewModelScope for ViewModels.  
     **Why:** Cancels coroutines automatically when lifecycle ends.
2. viewModelScope.launch(Dispatchers.IO) {
3. val data = repository.getData()
4. withContext(Dispatchers.Main) {
5. updateUI(data)
6. }
7. }
8. **Choose Correct Dispatcher**
   * Main → UI work
   * IO → Room DB, Retrofit calls
   * Default → Data processing, JSON parsing
   * Unconfined → Special/debug use cases
9. **Use withContext to Switch Threads**
10. val data = withContext(Dispatchers.IO) { apiCall() }
11. withContext(Dispatchers.Main) { updateUI(data) }
12. **Cancel Coroutines When Not Needed**
    * Avoid running coroutines longer than necessary.
    * Use isActive to check if coroutine is still running before heavy work.
13. **Avoid GlobalScope in Android**
    * GlobalScope coroutines live for the entire app life → **memory leaks risk**.
14. **Prefer Structured Concurrency**
    * Always launch coroutines in a **scope**.
    * Prevents “fire and forget” leaks.

**Part B — Extra & Future Knowledge**

**a. Exception Handling**

val handler = CoroutineExceptionHandler { \_, exception ->

println("Caught $exception")

}

launch(handler) {

throw RuntimeException("Boom!")

}

* Use try-catch inside launch/async or a CoroutineExceptionHandler.

**b. Coroutine Builders**

* launch → returns Job (no result)
* async → returns Deferred (result)
* runBlocking → blocking (use in main/test)
* withContext → switch threads

**c. Flow (Cold Async Streams)**

* Use Kotlin Flow for continuous data streams (e.g., Firebase updates).
* Supports operators like map, filter, collect.

**d. Timeout**

withTimeout(3000L) {

fetchData()

}

* Automatically cancels if exceeds time.

**e. Real-World Android Example**

* Fetch blockchain voting results in *Vote Chain* without blocking UI:

viewModelScope.launch(Dispatchers.IO) {

val results = blockchainApi.getVotes()

withContext(Dispatchers.Main) {

showResultsOnChart(results)

}

}