**Section 81 – Deep Dive into Coroutines**

**A. Key Concepts from the Section**

**1. What are Coroutines**

* **Definition:** Kotlin’s way of writing asynchronous code in a sequential, readable manner.
* **Purpose:** Perform long-running tasks **without blocking** the main/UI thread.
* **Lightweight Threads:** Cheaper to create than Java threads, so you can launch many without heavy memory cost.
* **Suspend/Resume:** Can pause execution (suspend) and resume later.

**2. Synchronous vs Asynchronous**

* **Synchronous:** Tasks execute one after another; blocks until finished.
* **Asynchronous:** Tasks execute independently; other tasks continue while one is running.

**3. Why Coroutines in Android**

* Main Thread (UI thread) handles:
  + Drawing views
  + Processing UI events
  + User interactions
* Blocking this thread → **UI freezes** → ANR (Application Not Responding) after ~5s.
* Coroutines solve this by:
  + Running work on background threads (e.g., Dispatchers.IO)
  + Updating UI only on main thread (Dispatchers.Main)

**4. Suspend Functions**

* Marked with suspend keyword.
* Can only be called:
  + From another suspend function
  + From inside a coroutine
* They **suspend the coroutine, not the thread**.
* Useful for long-running operations (network calls, file I/O).

**Example:**

suspend fun downloadBigFileFromNet() {

// Simulate long-running work

for (i in 1..1\_000\_000) {

// Example: log progress

Log.d("Download", "Progress: $i")

}

}

**5. Coroutine Scope**

* Defines **lifetime** of coroutines.
* Cancels all child coroutines when destroyed.
* Predefined scopes in Android:
  + **lifecycleScope** (Activity/Fragment lifecycle)
  + **viewModelScope** (ViewModel lifecycle)
* **GlobalScope:** Independent of lifecycle (generally avoid for UI-related tasks).

**Example:**

lifecycleScope.launch(Dispatchers.IO) {

// Background work

}

**6. Dispatchers**

* Decide **which thread** the coroutine runs on:
  + **Dispatchers.Main** → UI thread
  + **Dispatchers.IO** → Disk/Network I/O tasks
  + **Dispatchers.Default** → CPU-intensive work
* Switch contexts with withContext.

**Example:**

lifecycleScope.launch(Dispatchers.IO) {

val data = downloadData()

withContext(Dispatchers.Main) {

// Update UI

textView.text = data

}

}

**7. Coroutine Builders**

**launch**

* Returns Job (no result).
* Fire-and-forget tasks.

val job = lifecycleScope.launch {

// Do work

}

job.cancel() // Cancel if needed

**async**

* Returns Deferred<T> (future result).
* Must call .await() to get result.
* Good for **parallel execution**.

val result = async { fetchData() }.await()

**8. Sequential vs Parallel Execution**

* **Sequential:**
* val one = doTaskOne()
* val two = doTaskTwo()
* val sum = one + two
  + Runs one after the other.
* **Parallel:**
* val one = async { doTaskOne() }
* val two = async { doTaskTwo() }
* val sum = one.await() + two.await()
  + Runs simultaneously, reduces total time.

**9. withContext**

* Switch coroutine’s dispatcher **within the same coroutine**.
* Does not create a new coroutine like launch/async.

**B. Implementation Steps with Example Code**

**1. Setting up a Coroutine**

// 1. Import coroutines

import kotlinx.coroutines.\*

// 2. Launch coroutine in background

lifecycleScope.launch(Dispatchers.IO) {

val data = downloadBigFileFromNet() // Background task

// 3. Switch to main thread for UI update

withContext(Dispatchers.Main) {

textView.text = "Download Complete"

}

}

**Comments:**

* Dispatchers.IO → background thread for network/disk I/O.
* withContext(Dispatchers.Main) → required for UI updates.

**2. Suspend Function**

suspend fun downloadBigFileFromNet(): String {

delay(3000) // Simulates 3s download

return "File Content"

}

**Note:** delay() is non-blocking (doesn’t freeze UI).

**3. Parallel Execution Example**

lifecycleScope.launch {

val start = System.currentTimeMillis()

val result1 = async { doTaskOne() }

val result2 = async { doTaskTwo() }

val sum = result1.await() + result2.await()

Log.d("Result", "Sum: $sum in ${System.currentTimeMillis() - start}ms")

}

**4. Best Practices**

1. **Always Use Lifecycle-Aware Scopes**
   * **Why:** Prevents memory leaks & crashes when an Activity/Fragment is destroyed.
   * **What to use:**
     + lifecycleScope → For UI components.
     + viewModelScope → For ViewModels (persists through config changes but cancels when ViewModel is cleared).
   * **Example:**

kotlin

CopyEdit

viewModelScope.launch(Dispatchers.IO) {

val data = repo.getData()

withContext(Dispatchers.Main) {

updateUI(data)

}

}

1. **Structured Concurrency Over GlobalScope**
   * **Why:** GlobalScope coroutines live until the process ends — they ignore component lifecycles, which can cause:
     + Memory leaks
     + Updating destroyed views
     + Wasted network calls
   * Use structured scopes so that when a parent (Activity/Fragment/ViewModel) is cancelled, **all children coroutines are automatically cancelled**.
2. **Switch Contexts Only When Needed**
   * Heavy work → Dispatchers.IO or Dispatchers.Default
   * UI updates → Dispatchers.Main
   * Avoid switching too often, as it adds overhead.
3. **Avoid Blocking Code in Coroutines**
   * Never use Thread.sleep() or synchronous network calls inside coroutines — it defeats the purpose of non-blocking.
   * Use delay() for simulated waits.
   * Use suspend-enabled APIs for network or I/O.
4. **Parallelize Independent Tasks**
   * If tasks do not depend on each other, run them with async/await to save time.
   * Avoid running them sequentially unless dependency/order is required.
5. **Cancel Coroutines When Not Needed**
   * Store Job references and call .cancel() when appropriate.
   * In UI, cancellation avoids unnecessary work after navigation.
6. **Make Suspend Functions Cancellable**
   * Long loops should check isActive to exit early when coroutine is cancelled:

kotlin

CopyEdit

suspend fun processItems() {

for (item in items) {

if (!isActive) return

process(item)

}

}

1. **Error Handling**
   * Use try/catch in coroutines for expected exceptions.
   * Use CoroutineExceptionHandler for global exception handling.

**5. Tools & APIs**

* **Kotlin Coroutines Library**: kotlinx.coroutines
* **Dispatchers**: Main, IO, Default
* **Coroutine Builders**: launch, async
* **Context Switching**: withContext
* **Android Lifecycle Scopes**: lifecycleScope, viewModelScope

**Part B – Additional Important Knowledge (Deep Dive)**

**1. Exception Handling Patterns**

* **Local try/catch:**

kotlin

CopyEdit

lifecycleScope.launch {

try {

val data = fetchData()

} catch (e: IOException) {

showError(e.message)

}

}

* **Global handler:**

kotlin

CopyEdit

val handler = CoroutineExceptionHandler { \_, exception ->

Log.e("CoroutineError", "Caught: $exception")

}

lifecycleScope.launch(handler) { ... }

**2. SupervisorJob vs Job**

* **Job:** If one child fails, all children are cancelled.
* **SupervisorJob:** Children run independently; one child’s failure won’t cancel others.
* Used in scenarios where partial success is acceptable (e.g., fetching multiple APIs, showing whatever data loads).

**3. Flow – For Continuous/Reactive Streams**

* **Why:** Coroutines are great for one-shot tasks, but Flow handles **multiple emissions over time** (like LiveData but coroutine-friendly).
* **Example:**

kotlin

CopyEdit

viewModelScope.launch {

repo.getDataFlow().collect { data ->

updateUI(data)

}

}

**4. Testing Coroutines**

* Use **runTest** (modern replacement for runBlockingTest) to control virtual time in tests.
* Use TestCoroutineDispatcher to skip delays instantly.

**5. Structured Concurrency in Large Projects**

* Each layer of your app (UI, ViewModel, Repository) should own its coroutine scope.
* The repository should not run in the ViewModel’s scope — it should run in its own scope and return suspend functions or flows.

**6. Cancellation Safety**

* Always ensure suspend functions are **cooperative** — they must check for cancellation and release resources properly.
* Use finally blocks for cleanup:

kotlin

CopyEdit

try {

// work

} finally {

// cleanup even if cancelled

}

**7. Integration with Modern Android Architecture**

* Recommended pattern in industry:  
  **ViewModelScope + Coroutines + Flow + Repository pattern**
* UI triggers ViewModel → ViewModel calls suspend/flow functions from repository → Repository runs I/O in Dispatchers.IO → emits results back to UI.

**8. Performance Tips**

* Use Dispatchers.Default for CPU-heavy work instead of IO.
* Avoid launching thousands of coroutines unnecessarily — batch or debounce work.
* Keep coroutines short-lived when possible to free up resources.