1. Kotlin Coroutines (launch – async – withContext – Dispatcher)

Kotlin **Coroutines** là cách để xử lý bất đồng bộ rất nhẹ, mượt, tối ưu hơn Thread.

1.1 CoroutineScope + launch

launch = chạy coroutine không trả kết quả

```
GlobalScope.launch {
    println("Running in coroutine")
}
```

Đặc điểm:

- Không trả về kết quả
- Dùng cho: gọi API, insert DB, chạy background ngắn
- Non-blocking

> "launch dùng cho tác vụ không cần giá trị trả về, chạy bất đồng bộ không chặn thread."

1.2 async + await

async = chạy coroutine và trả về Deferred

→ `await()` để nhận kết quả

```
val result = async {
    2 + 3
}.await()

println(result) // 5
```

Dùng khi:

- Cần chạy song song
- Cần kết quả trả về
- Tối ưu API nhiều nguồn

> "async dùng cho tác vụ cần kết quả trả về, chạy song song và gom kết quả bằng await."

1.3 withContext

Chuyển thread trong coroutine

```
withContext(Dispatchers.IO) {
    fetchDataFromNetwork()
}
```

Dùng khi:

- Muốn chạy 1 block code ở thread khác
- Điều chỉnh thread đúng chuẩn Android
    - `Main` → UI
    - `IO` → API, DB
    - `Default` → CPU-heavy

1.4 Dispatcher (Main – IO – Default – Unconfined)

Dispatcher	Dùng khi	Ví dụ
Main	UI thread	update UI, click listener
IO	network + database	Retrofit, Room
Default	CPU-heavy	calculate, sorting
Unconfined	không khuyến nghị	testing

> "Dispatcher giúp phân luồng công việc: Main cho UI, IO cho network + database, Default cho xử lý nặng."

---

2. suspend function – Flow – StateFlow – SharedFlow

2.1 suspend function

Khái niệm

Hàm "suspend" là hàm chạy trong coroutine và **có thể tạm dừng mà không chặn thread**.

```
suspend fun fetchData(): String {
    delay(1000)
    return "Done"
}
```

> "Suspend function cho phép coroutine tạm dừng mà không block thread, dùng nhiều trong API call, DB query."

2.2 Flow (cold stream)

Flow là stream dữ liệu bất đồng bộ.

Đặc điểm:

- **Cold** → chỉ chạy khi có người collect
- Hỗ trợ operator như map, filter
- Tối ưu reactive UI

Ví dụ:

```
val flow = flow {
    emit(1)
    emit(2)
}
```

Collect:

```
flow.collect { println(it) }
```

2.3 StateFlow (hot stream – luôn có state)

Đặc điểm:

- Thuộc kotlinx.coroutines
- Luôn giữ giá trị cuối cùng (state)
- Rất phù hợp với ViewModel
- Không lifecycle-aware (cần collect trong lifecycleScope)

```
private val _count = MutableStateFlow(0)
val count = _count.asStateFlow()
```

2.4 SharedFlow (event stream)

Đặc điểm:

- Dùng cho event như: toast, navigation, click
- Không giữ state

- Phát lại nhiều subscriber

```
private val _event = MutableSharedFlow<String>()
val event = _event
```

2.5 Bảng so sánh Flow – StateFlow – SharedFlow

Loại	Có giá trị mặc định?	Hot/Cold	Use case
Flow	❌ Không	Cold	streaming, API
StateFlow	✔ Có	Hot	UI state, ViewModel
SharedFlow	❌ Không	Hot	Event (toast, navigate)

> "Flow là cold, StateFlow và SharedFlow là hot. StateFlow giữ state mới nhất, SharedFlow dùng cho event."

---

3. Extension function – Sealed class – Data class

3.1 Extension function

Khái niệm:

Thêm function vào class **mà không cần kế thừa**.

```
fun String.capitalizeFirst(): String {
    return this.replaceFirstChar { it.uppercase() }
}

println("hello".capitalizeFirst()) // Hello
```

> "Extension function giúp mở rộng class mà không cần modify source hoặc inheritance."

3.2 Data class

Khái niệm:

Class chuyên để chứa dữ liệu.

```
data class User(val name: String, val age: Int)
```

Tự tạo:

- equals
- hashCode
- toString
- copy()

3.3 Sealed class

Khái niệm:

Class giới hạn số lượng subclass → dùng để đại diện cho state UI.

Ví dụ về state API:

```
sealed class Result {
    object Loading : Result()
    data class Success(val data: String) : Result()
    data class Error(val msg: String) : Result()
}
```

> "Sealed class dùng để mô tả các trạng thái hữu hạn như Loading/Success/Error, rất phù hợp cho API UI state."

---

4. Null Safety – lateinit – lazy

4.1 Null Safety

Khái niệm:

Kotlin phân biệt rõ:

- `String` → không được null
- `String?` → có thể null

```
var name: String? = null
```

Operator quan trọng:

- `?.` → safe call
- `!!` → không null (cẩn thận crash)
- `?:` → default value (elvis)
- `let`
- `run`
- `apply`
- `also`

4.2 lateinit

Khái niệm:

Khai báo biến sẽ được gán giá trị sau (không phải null).

```
lateinit var binding: ActivityMainBinding
```

Lưu ý:

- Chỉ dùng với `var`
- Dùng cho object, không dùng cho primitive
- Nếu gọi trước khi init → crash lỗi *lateinit property not initialized*

4.3 lazy

Khái niệm:

Khởi tạo biến khi được dùng lần đầu.

```
val db by lazy { Room.databaseBuilder(...).build() }
```

Ưu điểm:

- Tối ưu performance
- Đảm bảo init 1 lần

Bảng so sánh:

Thuộc tính	lateinit	lazy
Dùng cho	var	val
Nullable?	Không	Không

Thuộc tính	lateinit	lazy
Khởi tạo lúc nào?	Gán sau	Khi dùng lần đầu
Crash nếu chưa init?	Có	Không

> "'lateinit' dùng cho var và init sau, còn 'lazy' dùng cho val và init khi dùng lần đầu."