**Rangkuman dari Berkenalan dengan Coroutines**

Di sub-modul ini, kita telah membahas tentang apa itu *concurrency*, *pararelism*, *threading*, bahkan penerapannya pada kotlin dengan menggunakan fitur **Coroutines**. Cukup menarik dan mudah ‘kan ? Oke, mari kita ingat - ingat kembali apa saja yang telah dipelajari hingga saat ini:

* **Concurrency**adalah beberapa proses yang terjadi pada suatu sistem. Terjadi apabila terdapat 2 (dua) atau lebih proses yang tumpang tindih dalam satu waktu.
* **Parallelism** sama seperti *concurrency*, namun 2 (dua) atau lebih proses tersebut dijalankan pada waktu yang sama persis.
* Ada beberapa konsep dasar dari penerapan concurrency yang harus dipelajari yaitu:
  + **Process**  
    Merupakan bagian dari proses aplikasi yang dijalankan. Setiap kali aplikasi dijalankan, maka saat itu juga proses dijalankan. Tergantung pada sistem operasi yang digunakan, suatu proses dapat terdiri dari beberapa thread yang menjalankan instruksi secara bersamaan.
  + **Thread**  
    Biasa dikenal sebagai proses yang ringan dan merupakan komponen dari suatu proses aplikasi yang menjalankan tugas tertentu secara spesifik.
  + **I/O-Bound**  
    Sebuah algoritma yang bergantung pada perangkat input atau output. Waktu untuk mengeksekusi sebuah I/O-bound tergantung pada kecepatan perangkat yang digunakan.
* Ada beberapa permasalahan yang ditimbulkan ketika menerapkan concurrency pada aplikasi, yaitu :
  + **Deadlocks**  
    Sebuah kondisi di mana dua proses atau lebih saling menunggu proses yang lain untuk melepaskan resource yang sedang digunakan yang mengakibatkan proses yang sedang berjalan tak kunjung selesai melakukan operasinya.
  + **Livelocks**  
    Kondisi di mana sebuah proses tidak bisa melanjutkan tugasnya, sama seperti **deadlocks**. Namun, perbedaannya adalah selama **livelocks** terjadi, keadaan aplikasi tetap bisa berubah. Walau perubahan keadaan tersebut menyebabkan proses berjalan dengan tidak semestinya.
  + **Starvation**  
    Kondisi yang biasa terjadi setelah **deadlock**. Kondisi ini sering kali menyebabkan sebuah proses kekurangan sumber daya. Pada kondisi ini, thread menjadi tidak mendapatkan akses reguler ke sumber daya bersama dan membuat proses terhenti.
  + **Race Conditions**Kondisi di mana terdapat banyak thread yang mengakses data yang dibagikan bersama dan mencoba mengubahnya secara bersamaan. Ini bisa terjadi ketika kode concurrent yang dituliskan untuk berjalan secara sekuensial.
* **Coroutines** merupakan fitur unggulan dan cara baru dalam menuliskan thread yang ringan dan efisien pada Kotlin. Disediakan JetBrains khusus untuk Kotlin.
* Untuk menjalankan coroutine, diperlukan fungsi yang biasa disebut sebagai **coroutine builder** yang mengambil suspending lambda dan membuat coroutine untuk menjalankannya. Ada beberapa macam *coroutine builder* yaitu:
  + **launch**  
    Fungsi ini digunakan untuk memulai sebuah coroutines yang tidak akan mengembalikan sebuah hasil dan menghasilkan Job yang dapat digunakan untuk membatalkan eksekusi.
  + **runBlocking**  
    Dibuat untuk menjembatani blocking code menjadi kode yang dapat ditangguhkan. runBlocking akan memblokir sebuah thread yang sedang berjalan hingga eksekusi coroutine selesai
  + **async**  
    Fungsi ini digunakan untuk memulai sebuah coroutine yang akan mengembalikan sebuah hasil.
* **Job**merupakan hasil dari perintah asynchronous yang dijalankan dan merepresentasikan coroutine yang sebenarnya. Memiliki 3 properti terdiri dari **isActive, isCompleted, dan isCancelled** yang menunjukkan *state* dari Job tersebut.
* **Deferred**adalah nilai kembalian dari fungsi **async** yang dapat berupa hasil atau exception. Merupakan nilai tangguhan yang dihasilkan dari proses coroutines dan bisa kita kelola sesuai dengan kebutuhan.
* **CoroutineDispatcher**, merupakan *base class* yang menentukan thread yang berfungsi untuk menjalankan coroutines. Class ini diimplementasikan pada beberapa class berikut :
  + **Dispatcher.Default**  
    Merupakan dispatcher dasar yang digunakan oleh semua standard builders seperti *launch, async*, dll jika tidak ada dispatcher lain yang ditentukan. Dispatcher.Default menggunakan kumpulan thread yang ada pada JVM. Jumlah maksimal thread yang digunakan adalah sama dengan jumlah core dari CPU.
  + **Dispatcher.IO**  
    Merupakan dispatcher yang digunakan untuk membongkar pemblokiran operasi I/O dan akan menggunakan kumpulan thread yang dibuat berdasarkan permintaan.
  + **Dispatcher.Unconfined**  
    Merupakan Dispatcher yang menjalankan coroutines pada thread yang sedang berjalan sampai mencapai titik penangguhan. Setelah penangguhan, coroutines akan dilanjutkan pada thread dimana komputasi penangguhan yang dipanggil.
* Selain dengan menggunakan metode diatas, ada beberapa *builder* lain yang dapat digunakan untuk menentukan *thread* yang dibutuhkan, antara lain:
  + **Single Thread Context**  
    Dispatcher ini menjamin bahwa setiap saat coroutine akan dijalankan pada thread yang ditentukan. Penerapannya dilakukan dengan memanfaatkan fungsi **newSingleThreadContext()** pada parameter dispatcher dalam fungsi **launch().**
  + **Thread Pool**  
    Sebuah dispatcher yang memiliki kumpulan thread. Ia akan memulai dan melanjutkan coroutine di salah satu thread yang tersedia pada kumpulan tersebut. Runtime akan menentukan thread mana yang tersedia dan juga menentukan bagaimana proses distribusi bebannya. Penerapannya dilakukan dengan memanfaatkan fungsi **newFixedThreadPoolContext()** pada parameter dispatcher dalam fungsi **launch()**.
* **Channel**adalah nilai deferred yang menyediakan cara mudah untuk mentransfer nilai tunggal antara coroutine atau bisa disebut melakukan interaksi antar coroutine serta membuat kode *concurrent* dapat berkomunikasi dengan mengirim dan menerima pesan tanpa harus peduli di thread mana coroutine berjalan.