# F.1 – Async Programming

## Concurrency e Threads

Finora abbiamo considerato l’app come se fosse una serie di eventi consecutivi **(sequential)**.   
Con l'aumentare della complessità della nostra app però, sarà necessario cominciare ad integrare il concetto di concorrenza (**concurrency)**, ovvero operazioni che si svolgono in “parallelo”.

Un’operazione si dice Concurrent quanto è in grado di essere eseguita fuori ordine o apparentemente in parallelo, consentendo un uso più efficiente delle risorse.

Il multitasking è l'esecuzione simultanea di più attività concorrenti durante un certo periodo di tempo.

Chart, bar chart

Description automatically generated

Utilizzare operazioni che avvengono in parallelo ci permetterà di evitare che l'app si blocchi nel caso dovessimo effettuare delle operazioni particolarmente pesanti, come l'elaborazione di immagini o la richiesta di dati da un servizio remoto. Queste saranno eseguite in background, mentre l’utente potrà continuare ad utilizzare l’interfaccia.

### Definizione

**I Thread** sono delle sequenze che spezzettano il nostro programma e sono eseguite in maniera concorrente.

Il sistema Android utilizza vari thread. Il Main thread è quello che viene utilizzato di base per tutto, inclusa l’interfaccia. Appesantire troppo questo thread rende l’interfaccia inutilizzabile, quindi a volte è necessario spostare le operazioni su altri thread.

Inoltre, Android pone dei limiti a cosa può essere fatto sul Main thread. Ad esempio, lanciare un’operazione di rete (*Network*) sul Main thread creerà un’eccezione.

Vediamo un esempio di concorrenza utilizzando il sito   
<https://developer.android.com/training/kotlinplayground>

Diagram, venn diagram

Description automatically generated

Utilizza questo codice, che crea tre Thread concorrenti:

fun main() {  
   val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")  
   repeat(3) {  
       Thread {  
           println("${Thread.currentThread()} has started")  
           for (i in states) {  
               println("${Thread.currentThread()} - $i")  
               Thread.sleep(50)  
           }  
       }.start()  
   }  
}

I thread saranno stampati in questa maniera:

Thread[Thread-0,5,main] – Message

All’interno delle parentesi c’è il nome del thread (Thread-n), la sua priorità (5), ed il suo gruppo (main)

Lanciandolo varie volte, vedrai che il risultato non sarà sempre lo stesso: in base alle risorse disponibili in quel momento, i thread verranno eseguiti in un’ordine non-specifico e non sequenziale.

I thread sono un modo di effettuare operazioni in parallelo, ma hanno varie problematiche da tenere in conto:

* Richiedono molte risorse, spesso nullificando lo scopo del multitasking.
* La mancanza di sicurezza dell’ordine con cui vengono eseguiti può produrre risultati imprevedibili.
* Più thread che cercano di accedere allo stesso momento alla stessa risorsa creano dei bug o crash (**race condition**)

Quindi è complicato lavorare direttamente con i thread. Per questo solitamente si utilizzano le Kotlin **Coroutines**

## Coroutines

https://kotlinlang.org/docs/coroutines-overview.html

### Definizione

Le Coroutines sono delle astrazioni costruite sulla base di Threads.   
Questo permette loro di essere **molto leggere,** di integrare features come la **possibilità di essere messe in pausa o interrotte**, e di lavorare in coro in un **multitasking cooperativo** tra i vari elementi concorrenti.

Una coroutine è composta da tre elementi principali:

|  |  |
| --- | --- |
| Job | Un'unità di lavoro cancellabile che racchiude delle operazioni, come quella creata con la funzione launch(). |
| CoroutineScope | È un *context* che impone regole ai suoi discendenti, come i criteri per la cancellazione. Le funzioni utilizzate per creare nuove coroutine come launch() e async() estendono CoroutineScope. |
| Dispatcher | Determina il thread che verrà utilizzato dalla coroutine.  Il dispatcher *Main* eseguirà sempre le coroutine sul Main thread, mentre i dispatcher come *Default*, *IO* o *Unconfined* utilizzeranno altri thread. |

## Suspend Functions

### Definizione

**suspend** è una keyword che indica che la funzione contiene un’altra suspend function. Una suspend function può essere invocata solo all’interno di una coroutine, o all’interno di un’altra suspend function.

Apri MoviesRepository e fai questa modifica:

A screen shot of a computer

Description automatically generated with low confidence

*delay()* è una funzione che mette in pausa la funzione dove è invocata, in maniera asincrona *non-blocking* .

Puoi vedere sulla sinistra che la **freccetta con l’intersezione verde** ti notifica che *su quella riga* c’è una *suspend function.*

Ora la funzione richederà circa 1 secondo per essere eseguita, e non potrà più essere eseguita in maniera sincrona, imponendo la concorrenza.

## viewModelScope, launch()

### Definizione viewModelScope

*viewModelScope* è un CoroutineScope molto importante fornito da ViewModel, che si occupa in maniera automatica di alcune attività. Se dovessimo lanciare delle funzioni che possono essere eseguite solo su alcuni specifici thread (ad esempio una richiesta network su Dispatchers.IO, o l’invio di dati all’interfaccia su Dispatchers.Main), *viewModelScope* lavorerebbe per noi effettuando questi salti da un thread all’altro.

Il lifecycle di viewModelScope è legato a quello del suo ViewModel: se questo cessa di esistere, anche lo scope si chiude, interrompendo tutte le coroutine in corso.

In un ViewModel:

A screen shot of a computer program

Description automatically generated with low confidence

*onCompleteListener* è una funzione anonima che passiamo come parametro, che ci permetterà di ottenere il risultato da una operazione asincrona, quando la invocheremo dalla view:

A picture containing text, screenshot, font

Description automatically generated

### Definizione launch()

*.launch* è una funzione che crea una coroutine.

Il contenuto viene eseguito in maniera asincrona, senza bloccare il Main thread: un particolare che ci viene specificato dalla documentazione.

*.launch* racchiude il suo contenuto in un Job, e produce questo Job come risultato (e non il suo contenuto). Quindi dobbiamo avere un altro modo per comunicare il risultato delle operazioni al suo interno. Nel nostro caso, è il listener.

## Futures

Finora abbiamo utilizzato una coroutine che permette di lanciare un pezzo di codice pesante in parallelo, permettendo al codice principale di continuare l’esecuzione.

Ci sono casi però in cui, all’interno di una coroutine, ci troviamo a dover lanciare piu’ suspend fun in parallelo.

Tipicamente questo avviene quando abbiamo bisogno non di una, ma di piu’ chiamate network. Inoltre, queste chiamate devono dare il risultato tutte insieme, perché aggregano i dati che singolarmente non avrebbero senso.

Un esempio: per mostrare il costo di un oggetto in vendita, dobbiamo ottenere il costo dell’oggetto da un servizio, e al contempo dobbiamo ottenere da un altro servizio il tasso di cambio tra la valuta originale dell’oggetto e quella dell’utente che lo sta comprando. Non avrebbe senso mostrare il prezzo in rupie a un utente europeo, e tantomeno avrebbe senso mostrare solo il tasso di cambio senza il prezzo dell’oggetto.

Una soluzione è lanciare all’interno della coroutine altre coroutines: è un metodo lecito e spesso utilizzato. Il problema però è che le coroutines non si coordineranno tra di loro; quindi, dovremmo inventarci un metodo per controllare ciclicamente i due risultati dell’esempio finchè entrambi non sono disponibili.

Per risolvere questo scomodo problema Kotlin ci viene in soccorso implementando il concetto di **Futures**

Un Future è un oggetto che funge da placeholder per un risultato inizialmente sconosciuto, di solito perché il calcolo del suo valore non è ancora completo.

<https://en.wikipedia.org/wiki/Futures_and_promises>

### Async / Await

Kotlin utilizza le funzioni .**async** per lanciare una coroutine che ritorna subito un future,   
e .**await** per utilizzare il future appena sarà disponibile.

Kotlin wrappa i futures in un oggetto chiamato **Deferred**, che offre alcune facilitazioni e funziona similmente a *Job*: ad esempio, se necessario, può interrompere la coroutine lanciata da *async* prima che sia terminata.

Deferred è una **Promise**. Il Future propriamente detto è il valore “wrappato” da Deferred.

A screen shot of a computer code

Description automatically generated with low confidence

* In questo esempio *getItemOriginalCost* o *getConversionRates* sono due suspend fun, e non sappiamo quale verrà risolta prima*.* Decidiamo quindi di lanciarle come Futures tramite *async*.
* *Async* genera delle coroutines e ritorna subito (sincronicamente) un Deferred.
* *originalCost* e *conversionRates* sono due Deferred. Entrambi wrappano i risultati Future delle funzioni nelle coroutines, che saranno dei Double.
* *convertCostToEuros* viene invocato dandogli come parametri i valori Double prodotti da *await,* e verrà effettivamente lanciato solo quando entrambi i valori saranno disponibili.
* Gli a*wait* fanno in modo di coordinare le due coroutines: quando arriva la prima, il processo aspetterà che arrivi anche l’altra prima di proseguire.
* Puoi vedere infatti che l’unica riga che blocca il processo è proprio questa, visualizzata con una freccetta con intersezione verde, sulla sinistra. Sono gli *await* a bloccare.

## Esercizio

Aggiungi a Movie Master:

* In MovieListViewModel:
  + ora *getMoviesResponse* nel repository non puo’ essere piu’ invocato in maniera sincrona. Modifica *getMovies* in maniera che non crei errore, e rendila **privata**.
  + crea una nuova funzione **pubblica** *getMoviesAsync* dove
    - Lancia una coroutine dove ottieni la lista di film
    - Rendi questo risultato ottenibile dalla view
    - Salva questo risfunultato in una variabile del viewModel
  + Fa in modo che *getMoviesById* resti una funzione sincrona, che prende i dati da una variabile nel quale è stato salvato il risultato della richiesta dei film al repository.
* In MovieListFragment:
  + Assegna l’adapter alla recyclerview con un dataset vuoto
  + Quando i dati dei film sono disponibili, aggiorna il dataset dell’adapter.
  + Aggiungi una *Progressbar* al layout. Deve mostrare un’animazione circolare di caricamento.
  + Quando i dati dei film non sono disponibili, Progressbar deve essere visibile, mentre Recyclerview deve essere invisibile
  + Quando i dati sono disponibili, RV è visibile e Progressbar invisibile.
* Esercizio Async/Await
* Nella pagina dove è presente la lista di attori\attrici
  + Prendi l’endpoint Images e mostra almeno 3 poster
  + Prendi l’endpoint Reviews e mostra almeno 3 reviews
* Lancia queste richieste in parallelo