Coroutines

Lezione 10

Introduzione

- Le coroutines sono un *concurrency design pattern* che si può usare su Android per semplificare il codice che viene eseguito in modo asincrono
- Sono state introdotte nella versione 13 di Kotlin
- In Android, le coroutine aiutano a gestire task di lunga durata (long-running tasks) che potrebbero altrimenti bloccare il thread principale e causare la mancata risposta dell'app
- Oltre il 50% degli sviluppatori professionisti che utilizzano le coroutine ha riferito di aver visto un aumento della produttività!

Kotlin le definisce..

- Una coroutine è un'istanza di calcolo sospendibile (suspendable computation)
- È concettualmente simile a un thread, nel senso che richiede l'esecuzione di un blocco di codice che funziona in concomitanza con il resto del codice
- Tuttavia, una coroutine non è legata a nessun particolare thread
  - Può sospendere la sua esecuzione in un thread e riprenderla in un altro.

Funzionalità

- Coroutines è la soluzione consigliata per la programmazione asincrona su Android!
- Le caratteristiche più rilevanti sono:
  - **Leggero**: si possono eseguire molte coroutine su un singolo thread grazie al supporto per la «sospensione» (*suspension*), che non blocca il thread in cui è in esecuzione la coroutine. La sospensione consente di risparmiare memoria rispetto al blocco supportando al tempo stesso molte operazioni simultanee
  - **Meno perdite di memoria**: usa la concorrenza strutturata (*structured concurrency*) per eseguire operazioni all'interno di un ambito (scope)
  - **Supporto di annullamento integrato**: l'annullamento (Cancellation) viene propagato automaticamente attraverso la gerarchia coroutine in esecuzione
  - **Integrazione con Jetpack**: molte librerie Jetpack includono estensioni che forniscono un supporto completo per le coroutine. Alcune librerie forniscono anche il proprio ambito di coroutine che è possibile utilizzare per la concorrenza strutturata.

Approfondimento: structured concurrency

- Le coroutine seguono un principio di concorrenza strutturata (*structured concurrency*), il che significa che le nuove coroutine possono essere eseguite solo in uno specifico CoroutineScope che delimita la durata della coroutine
- In una «vera» applicazione, lancerai molte coroutine
  - La concorrenza strutturata assicura che non vadano persi e non «sfuggano». Un ambito (scope) esterno non può essere completato finché tutte le sue coroutine figli non sono state completate. La concorrenza strutturata garantisce inoltre che eventuali errori nel codice vengano segnalati correttamente e non vadano mai persi

Per capire meglio..

- Vediamo un esempio!

- Vogliamo effettuare una richiesta di rete e restituire il risultato al thread principale, dove l'app può quindi visualizzare il risultato all'utente
- In particolare, il componente **ViewModel** chiama il livello di **repository** sul thread principale per attivare la richiesta di rete
- Noi vogliamo mantenere sbloccato il thread principale
- Nota: ViewModel include un set di estensioni KTX che funzionano direttamente con le coroutine
  - Queste estensioni sono librerie lifecycle-viewmodel-ktx

Dependency info

Prima di tutto, aggiungiamo la dipendenza in build.gradle file:

```
dependencies {
    implementation("org.jetbrains.kotlinx:ko
tlinx-coroutines-android:1.3.9")
}
```

Esempio

- Effettuare una richiesta di rete sul thread principale lo fa attendere o bloccare fino a quando non riceve una risposta
- Poiché il thread è bloccato, il sistema operativo non è in grado di disegnare l'interfaccia, il che causa il blocco dell'app e potenzialmente porta a una finestra di dialogo «Application Not Responding»
- Per una migliore esperienza utente, bisogna eseguire l'operazione su un thread in background

Esempio

```
class LoginRepository(private val responseParser: LoginResponseParser) {
    private const val loginUrl = "https://example.com/login"

    // Function that makes the network request, blocking the current thread
    fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {
        val url = URL(loginUrl)
        (url.openConnection() as? HttpURLConnection)?.run {
            requestMethod = "POST"
            setRequestProperty("Content-Type", "application/json; utf-8")
            setRequestProperty("Accept", "application/json")
            doOutput = true
            outputStream.write(jsonBody.toByteArray())
            return Result.Success(responseParser.parse(inputStream))
        }
        return Result.Error(Exception("Cannot open HttpURLConnection"))
    }
}
```

Esempio

- Il ViewModel attiva la richiesta di rete quando l'utente fa clic, ad esempio, su un pulsante:

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {
        val jsonBody = "{ username: \"$username\", token: \"$token\"}"
        loginRepository.makeLoginRequest(jsonBody)
            //makeLoginRequest poi fa effettivamente la chiamata di rete
    }
}
```

Esempio

- Con il codice precedente, LoginViewModel blocca il thread dell'interfaccia utente durante la richiesta di rete
- La soluzione più semplice per spostare l'esecuzione fuori dal thread principale è creare una nuova coroutine ed eseguire la richiesta di rete su un thread I/O
- Ovvero..

Esempio

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {
        // Create a new coroutine to move the execution off the UI thread
        viewModelScope.launch(Dispatchers.IO) {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"
            loginRepository.makeLoginRequest(jsonBody)
        }
    }
}
```

Esempio

- Ovvero:
  - viewModelScope è un CoroutineScope predefinito incluso nelle estensioni ViewModel KTX
    - Si noti che tutte le coroutine devono essere eseguite in un ambito (scope).
    - Un CoroutineScope gestisce una o più coroutine correlate
  - launch è una funzione che crea una coroutine e invia l'esecuzione del corpo della sua funzione al dispatcher corrispondente
  - Dispatchers.IO indica che questa coroutine deve essere eseguita su un thread riservato alle operazioni di I/O

Ricapitolando

- La funzione *login* viene eseguita come segue:
  - L'app chiama la funzione di login dal View layer nel thread principale
  - *launch* crea una nuova coroutine e la richiesta di rete viene effettuata in modo indipendente su un thread riservato alle operazioni di I/O
  - Mentre la coroutine è in esecuzione, la funzione login continua l'esecuzione e ritorna, probabilmente prima che la richiesta di rete sia terminata

Altro problema..

- Poiché questa coroutine viene avviata con viewModelScope, viene eseguita nell'ambito di ViewModel
- Se il ViewModel viene distrutto perché l'utente naviga via dal quello screen, viewModelScope viene annullato automaticamente e vengono annullate anche tutte le coroutine in esecuzione
- Un problema con l'esempio precedente è che qualsiasi cosa chiami makeLoginRequest deve ricordarsi di spostare esplicitamente l'esecuzione fuori dal thread principale
- Vediamo come possiamo modificare il repository per risolvere questo problema

Usare coroutines per main-safety

- Una funzione si definisce *main-safe* quando non blocca gli aggiornamenti dell'interfaccia utente sul thread principale
- La funzione makeLoginRequest NON è main-safe, poiché la chiamata a makeLoginRequest dal thread principale blocca l'interfaccia utente
- SOLUZIONE: Utilizzare la funzione *withContext()* della libreria coroutine per spostare l'esecuzione di una coroutine in un thread diverso

```
class LoginRepository(...) {
    ...
    suspend fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {

        // Move the execution of the coroutine to the I/O dispatcher
        return withContext(Dispatchers.IO) {
            // Blocking network request code
        }
    }
}
```

Quindi..

- *withContext(Dispatchers.IO)* sposta l'esecuzione della coroutine in un thread I/O, rendendo la nostra funzione di chiamata main-safe e consentendo all'interfaccia utente di aggiornarsi secondo necessità
- makeLoginRequest è anche contrassegnato con la parola chiave suspend
  - Questa parola chiave è il modo di Kotlin per «imporre» la chiamata di una funzione all'interno di una coroutine

- A questo punto però…
- Poiché makeLoginRequest sposta l'esecuzione fuori dal thread principale => la coroutine nella funzione login può ora essere eseguita nel thread principale

- class LoginViewModel(
  private val loginRepository: LoginRepository
  ): ViewModel() {

  fun login(username: String, token: String) {

  // Create a new coroutine on the UI thread
  viewModelScope.launch { //abbiamo tolto (Dispatchers.IO)
  val jsonBody = "{ username: \"$username\", token: \"$token\"}"

  // Make the network call and suspend execution until it finishes
  val result = loginRepository.makeLoginRequest(jsonBody)

  // Display result of the network request to the user
  when (result) {
  is Result.Success<LoginResponse> -> // Happy path
  else -> // Show error in UI
  }
  }
  }
  }

NOTA

- Da notare che la coroutine è ancora necessaria poiché makeLoginRequest è una funzione «suspended» e tutte le funzioni suspended devono essere eseguite in una coroutine
- Questo codice differisce dal precedente esempio di accesso in un paio di modi:
  - launch non accetta un parametro Dispatchers.IO. Quando non si passa un Dispatcher per l'avvio, tutte le coroutine avviate da viewModelScope vengono eseguite nel thread principale
  - Il risultato della richiesta di rete viene ora gestito per visualizzare la UI di successo o il fallimento
- La funzione di login ora viene eseguita come segue:
  - L'app chiama la funzione login() dal livello View sul thread principale
  - launch crea una nuova coroutine sul thread principale e la coroutine inizia l'esecuzione.
  - All'interno della coroutine, la chiamata a loginRepository.makeLoginRequest() ora sospende ulteriori esecuzioni della coroutine fino a quando il blocco withContext in makeLoginRequest() non termina l'esecuzione
  - Una volta terminato il blocco withContext, la coroutine in login() riprende l'esecuzione sul thread principale con il risultato della richiesta di rete

Domanda?

- Come facciamo a comunicare tra ViewModel e View?

Domanda?

- Come facciamo a comunicare tra ViewModel e View?
  - StateFlow!

E se vogliamo gestire le eccezioni!

- Solito! Try – Catch

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun makeLoginRequest(username: String, token: String) {
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"
            val result = try {
                loginRepository.makeLoginRequest(jsonBody)
            } catch(e: Exception) {
                Result.Error(Exception("Network request failed"))
            }
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

E se vogliamo gestire le eccezioni!

- Solito! Try – Catch

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun makeLoginRequest(username: String, token: String) {
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"
            val result = try {
                loginRepository.makeLoginRequest(jsonBody)
            } catch(e: Exception) {
                Result.Error(Exception("Network request failed"))
            }
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

Usa le coroutine per la main-safety

- Le coroutine Kotlin utilizzano i **dispatcher** per determinare quali thread vengono utilizzati per l'esecuzione delle coroutine
- Per eseguire codice al di fuori del thread principale, puoi dire alle coroutine di Kotlin di eseguire il lavoro sul dispatcher predefinito o IO
- In Kotlin, tutte le coroutine devono essere eseguite in un dispatcher, anche quando sono in esecuzione sul thread principale
- Le coroutine possono «auto-sospendersi» e il dispatcher è responsabile della loro ripresa

Dispatcher

- Per specificare dove devono essere eseguite le coroutine, Kotlin fornisce tre dispatcher che puoi utilizzare:
  - **Dispatchers.Main**: usare questo dispatcher per eseguire una coroutine sul thread Android principale. Dovrebbe essere usato solo per interagire con l'interfaccia utente ed eseguire un lavoro rapido. Gli esempi includono la chiamata di funzioni di sospensione, l'esecuzione di operazioni del framework UI e l'aggiornamento dell'interfaccia
  - **Dispatchers.IO**: questo dispatcher è ottimizzato per eseguire I/O su disco o rete al di fuori del thread principale. Gli esempi includono l'utilizzo del componente Room, la lettura o la scrittura su file e l'esecuzione di qualsiasi operazione di rete
  - **Dispatchers.Default**: questo dispatcher è ottimizzato per eseguire operazioni a uso intensivo della CPU al di fuori del thread principale. Esempi di casi d'uso includono l'ordinamento di un elenco e l'analisi di JSON.

Main-safe

- Con le coroutine, si può avviare thread con un controllo granulare
- Poiché withContext() consente di controllare il pool di thread di qualsiasi riga di codice senza introdurre callback, si può applicarlo a funzioni molto piccole come la lettura da un database o l'esecuzione di una richiesta di rete
- Una buona pratica è utilizzare withContext() per assicurarsi che ogni funzione sia main-safe, il che significa che è possibile chiamare la funzione dal thread principale
  - In questo modo, il chiamante non deve mai pensare a quale thread utilizzare per eseguire la funzione

Nota!

- L'uso di «suspend» non dice a Kotlin di eseguire una funzione su un thread in background
- È normale che le funzioni «suspend» operino sul thread principale
- È anche comune lanciare coroutine sul thread principale
- Dovresti sempre usare withContext() all'interno di una funzione «suspend» se hai bisogno di main-safety
  - come quando leggi o scrivi su disco, esegui operazioni di rete o esegui operazioni ad alta intensità di CPU

Start a coroutine

- Puoi avviare le coroutine in due modi:
  - *launch* avvia una nuova coroutine e non restituisce il risultato al chiamante. Qualsiasi task del quale non vi dovete preoccupare (in inglese si dice «fire and forgot») può essere avviato utilizzando launch
  - *async* avvia una nuova coroutine e consente di restituire un risultato con una funzione di sospensione chiamata *await*
- In genere, dovresti usare *launch* per iniziare una nuova coroutine da una funzione normale, poiché una funzione normale non può chiamare await
- Usa async solo quando sei all'interno di un'altra coroutine o quando sei all'interno di una funzione di sospensione (suspend) o quando esegui parallel decomposition

WARNING!

- *launch* e *async* gestiscono le eccezioni in modo diverso
- Poiché async prevede un'eventuale chiamata a *await*, contiene le eccezioni e le genera nuovamente come parte della chiamata in attesa
  - Ciò significa che se usi async per avviare una nuova coroutine da una funzione normale, potresti eliminare silenziosamente un'eccezione. Queste eccezioni eliminate non verranno visualizzate nelle metriche sugli arresti anomali né verranno annotate in logcat.
  - Per ulteriori informazioni, consulta Annullamento ed eccezioni nelle coroutine

Parallel decomposition

- Tutte le coroutine che vengono avviate all'interno di una funzione sospend devono essere stoppate quando tale funzione ritorna, quindi è probabile che tu debba garantire che quelle coroutine finiscano prima di tornare
- Con parallel decomposition in Kotlin, puoi definire un coroutineScope che avvia una o più coroutine
- Quindi, utilizzando await() (per una singola coroutine) o awaitAll() (per più coroutine), puoi garantire che queste coroutine finiscano prima di tornare dalla funzione

Esempio

- Definiamo un coroutineScope che recupera due documenti in modo asincrono; chiamando await() su ogni riferimento differito, garantiamo che entrambe le operazioni asincrone finiscano prima di restituire un valore:

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

Esempio

- Puoi anche utilizzare awaitAll() sulla collection, come mostrato nell'esempio seguente:

```
suspend fun fetchTwoDocs() =       // called on any Dispatcher (any thread, possibly Main)
    coroutineScope {
        val deferreds = listOf(     // fetch two docs at the same time
            async { fetchDoc(1) },  // async returns a result for the first doc
            async { fetchDoc(2) }   // async returns a result for the second doc
        )
        deferreds.awaitAll()        // use awaitAll to wait for both network requests
    }
```

Note

- Anche se fetchTwoDocs() avvia nuove coroutine con async, la funzione utilizza awaitAll() per attendere che le coroutine lanciate finiscano prima di tornare
- NOTA: anche se non avessimo chiamato awaitAll(), il builder coroutineScope non riprenderà la coroutine che ha chiamato fetchTwoDocs finché tutte le nuove coroutine non saranno state completate
- Inoltre, coroutineScope rileva eventuali eccezioni lanciate dalle coroutine e le reindirizza al chiamante
  - Per ulteriori informazioni Composing suspending functions

Concetti: CoroutineScope

- Un CoroutineScope tiene traccia di qualsiasi coroutine che crea utilizzando launch o async
- La coroutine in esecuzione può essere annullata chiamando scope.cancel() in qualsiasi momento
- In Android, alcune librerie KTX forniscono il proprio CoroutineScope per determinate classi del ciclo di vita
  - Ad esempio, ViewModel ha un **viewModelScope** e Lifecycle ha **lifecycleScope**
- A differenza di un dispatcher, tuttavia, un CoroutineScope non esegue le coroutine

CoroutineScope

- Se devi creare il tuo CoroutineScope per controllare il ciclo di vita delle coroutine in un particolare livello della tua app, puoi crearne uno come segue:

```
class ExampleClass {

    // Job and Dispatcher are combined into a CoroutineContext which
    // will be discussed shortly
    val scope = CoroutineScope(Job() + Dispatchers.Main)

    fun exampleMethod() {
        // Starts a new coroutine within the scope
        scope.launch {
            // New coroutine that can call suspend functions
            fetchDocs()
        }
    }

    fun cleanUp() {
        // Cancel the scope to cancel ongoing coroutines work
        scope.cancel()
    }
}
```

Un scope annullato/cancel non può creare più coroutine. Pertanto, dovresti chiamare scope.cancel() solo quando la classe che controlla il suo ciclo di vita viene distrutta. Quando si utilizza viewModelScope, la classe ViewModel annulla automaticamente lo scope nel metodo onCleared() di ViewModel.

Concetti: Job

- Un Job è un handle per una coroutine
- Ogni coroutine che crei con launch o async restituisce un'istanza di job che identifica in modo univoco la coroutine e ne gestisce il ciclo di vita
- Puoi anche passare un Job a un CoroutineScope per gestirne ulteriormente il ciclo di vita, come mostrato nell'esempio nella prossima slide

Job

```
class ExampleClass {
    ...
    fun exampleMethod() {
        // Handle to the coroutine, you can control its lifecycle
        val job = scope.launch {
            // New coroutine
        }

        if (...) {
            // Cancel the coroutine started above, this doesn't affect the scope
            // this coroutine was launched in
            job.cancel()
        }
    }
}
```

In un'applicazione potresti aver bisogno di un controllo granulare sulle tue coroutine in background. Ad esempio, un utente potrebbe aver chiuso la pagina che ha lanciato una coroutine e ora il suo risultato non è più necessario e la sua operazione può essere annullata. La funzione launch restituisce un Job che può essere utilizzato per annullare la coroutine in esecuzione

Concetti: CoroutineContext

- Un CoroutineContext definisce il comportamento di una coroutine utilizzando il seguente set di elementi:
  - Jon: controlla il ciclo di vita della coroutine
  - CoroutineDispatcher: invia il lavoro al thread appropriato
  - CoroutineName: il nome della coroutine, utile per il debug
  - CoroutineExceptionHandler: gestisce le eccezioni non rilevate
- Per le nuove coroutine create all'interno di un ambito (scope), una nuova istanza Job viene assegnata alla nuova coroutine e gli altri elementi CoroutineContext vengono ereditati dall'ambito (scope) che li contiene
  - Puoi eseguire l'override degli elementi ereditati passando un nuovo CoroutineContext alla funzione launch o async. Si noti che il passaggio di un Job a launch o async non ha alcun effetto, poiché una nuova istanza di Job viene sempre assegnata a una nuova coroutine

Esempio

```
class ExampleClass {
    val scope = CoroutineScope(Job() + Dispatchers.Main)

    fun exampleMethod() {
        // Starts a new coroutine on Dispatchers.Main as it's the scope's default
        val job1 = scope.launch {
            // New coroutine with CoroutineName = "coroutine" (default)
        }

        // Starts a new coroutine on Dispatchers.Default
        val job2 = scope.launch(Dispatchers.Default + CoroutineName("BackgroundCoroutine")) {
            // New coroutine with CoroutineName = "BackgroundCoroutine" (overridden)
        }
    }
}
```

e.. Flow?

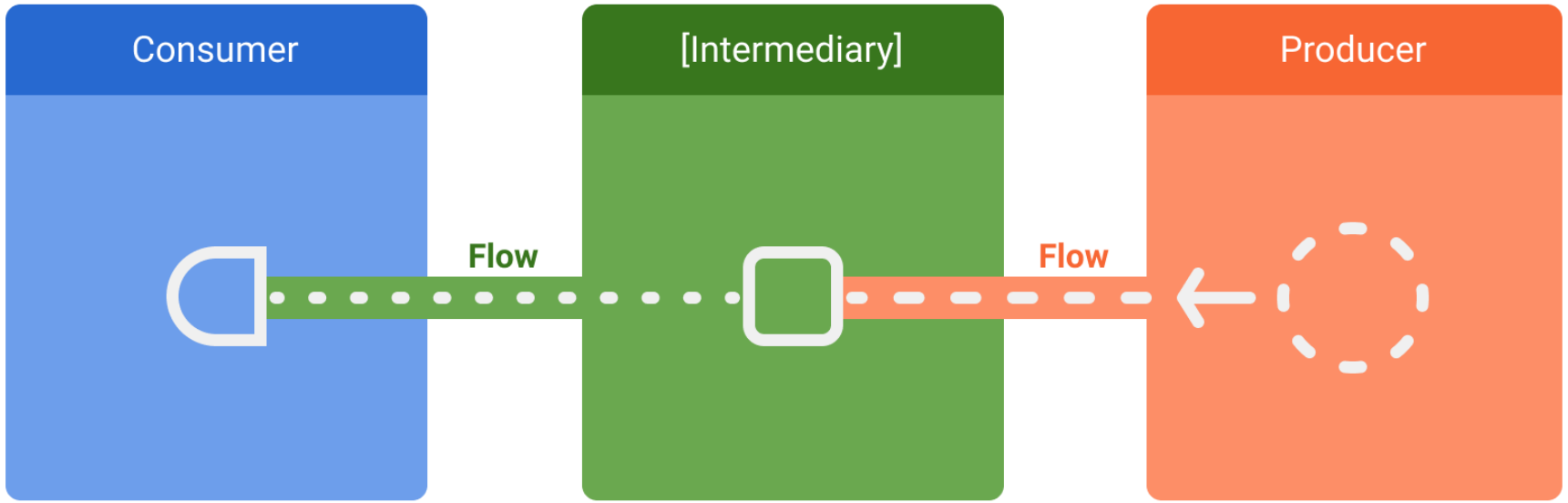
- Nelle coroutine, un Flow è un tipo che può emettere più valori in sequenza, al contrario delle *suspend function* che restituiscono un solo valore
  - È quindi una versione asincrona di un *Sequence*, un tipo di collection
    - In realtà, a differenza delle collection, le sequence non contengono elementi, li producono durante l'iterazione
- Ad esempio, si può utilizzare un flow per ricevere aggiornamenti in tempo reale da un database
- I flow sono costruiti sopra le coroutine e possono fornire più valori
- Un flow è concettualmente un flusso di dati che può essere calcolato (computed) in modo asincrono
- I valori emessi devono essere dello stesso tipo
  - Ad esempio, un Flow<Int> è un flusso che emette valori interi

Flow

- Un flow è molto simile a un Iterator che produce una sequenza di valori, ma utilizza funzioni di sospensione (suspend funtion) per produce e consume valori in modo asincrono
  - Ciò significa, ad esempio, che il flusso può effettuare in modo sicuro una richiesta di rete per produrre il valore successivo senza bloccare il thread principale
- Ci sono tre entità coinvolte nei flussi di dati:
  - Un **producer** produce dati che vengono aggiunti al flusso. Grazie alle coroutine, i flow possono anche produrre dati in modo asincrono
  - (Facoltativo) Gli **intermediari** possono modificare ciascun valore emesso nel flusso o il flusso stesso
  - Un **consumer** consuma i valori dal flusso

Flow

Entità coinvolte in un flusso di dati



Creare un Flow

- Per creare flussi, utilizza flow builder API
- La flow builder function crea un nuovo flusso in cui è possibile emettere manualmente nuovi valori nel flusso di dati utilizzando la funzione *emit*
- Nell'esempio seguente, la data source recupera automaticamente le notizie più recenti a intervalli fissi
  - Poiché una funzione di sospensione (suspend function) non può restituire più valori consecutivi, la data source restituisce un flow per soddisfare questo requisito. In questo caso, la data source funge da produttore

Esempio: Flow

- ```
  class NewsRemoteDataSource(
      private val newsApi: NewsApi,
      private val refreshIntervalMs: Long = 5000
  ) {
      val latestNews: Flow<List<ArticleHeadline>> = flow {
          while(true) {
              val latestNews = newsApi.fetchLatestNews()
              emit(latestNews) // Emits the result of the request to the flow
              delay(refreshIntervalMs) // Suspends the coroutine for some time
          }
      }
  }

  // Interface that provides a way to make network requests with suspend functions
  interface NewsApi {
      suspend fun fetchLatestNews(): List<ArticleHeadline>
  }
  ```
  ```

Flow

- Il flusso builder viene eseguito all'interno di una coroutine
- Pertanto, beneficia delle stesse API asincrone, ma si applicano alcune restrizioni:
  - I flow sono sequenziali. Poiché il produttore si trova in una coroutine, quando chiama una funzione di sospensione, il produttore sospende fino al ritorno della funzione di sospensione
    - Nell'esempio, il produttore sospende fino al completamento della richiesta di rete fetchLatestNews. Solo allora il risultato viene emesso nel flusso.
  - Con il flow builder, il producer non può emettere valori da un CoroutineContext diverso. Pertanto, non chiamare emit in un CoroutineContext diverso creando nuove coroutine o utilizzando blocchi di codice withContext. In questi casi puoi utilizzare altri generatori di flussi come callbackFlow

Collecting da un Flow

- Per ottenere tutti i valori nel flusso man mano che vengono emessi, usa *collect*
- Poiché *collect* è una suspend function, deve essere eseguita all'interno di una coroutine
- Prende un lambda come parametro che viene chiamato su ogni nuovo valore
- Poiché si tratta di una suspend function, la coroutine che chiama collect può sospendere finché il flusso non viene chiuso
- Vediamo ora una semplice implementazione di un ViewModel che consuma i dati dal livello del repository

Collecting da un Flow

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {

    init {
        viewModelScope.launch {
            // Trigger the flow and consume its elements using collect
            newsRepository.favoriteLatestNews.collect { favoriteNews ->
                // Update View with the latest favorite news
            }
        }
    }
}
```

Flows in Jetpack libraries

- Flow è integrato in molte librerie Jetpack ed è popolare tra le librerie Android di terze parti
- Flow è perfetto per gli aggiornamenti dei dati in tempo reale e flussi infiniti di dati
- Si può utilizzare Flow con Room per essere avvisato delle modifiche in un database
  - Quando si usano oggetti di accesso ai dati (DAO), restituire un tipo di flusso per ottenere aggiornamenti in tempo reale

Ma quindi.. StateFlow?

- StateFlow è un API di Flow che consente ai flussi di emettere in modo ottimale aggiornamenti di stato ed emettere valori a più consumatori

StateFlow (Ripasso..)

- StateFlow è un data holder observable flow che emette gli aggiornamenti di stato correnti e nuovi
- La sua proprietà value riflette il valore dello stato corrente
- Per aggiornare lo stato e inviarlo al flusso, occorre assegnare un nuovo valore alla proprietà value della classe MutableStateFlow
- Uno StateFlow può essere esposto da MyClasseUiState in modo che i componenti Composable possano ascoltare gli aggiornamenti dello stato dell'interfaccia utente e fare in modo che lo stato dello schermo sopravviva alle modifiche della configurazione

- Rivediamo l’ultimo esempio ma con StateFlow

Esempio StateFlow

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {

    // Backing property to avoid state updates from other classes
    private val _uiState = MutableStateFlow(LatestNewsUiState.Success(emptyList()))
    // The UI collects from this StateFlow to get its state updates
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    init {
        viewModelScope.launch {
            newsRepository.favoriteLatestNews
                // Update View with the latest favorite news
                // Writes to the value property of MutableStateFlow,
                // adding a new element to the flow and updating all
                // of its collectors
                .collect { favoriteNews ->
                    _uiState.value = LatestNewsUiState.Success(favoriteNews)
                }
        }
    }
}

// Represents different states for the LatestNews screen
sealed class LatestNewsUiState {
    data class Success(val news: List<ArticleHeadline>): LatestNewsUiState()
    data class Error(val exception: Throwable): LatestNewsUiState()
}
```

StateFlow

- La classe responsabile dell'aggiornamento di un MutableStateFlow è il producer e tutte le classi che raccolgono da StateFlow sono i consumer
- A differenza di un flow creato utilizzando il generatore di flussi (flow builder), con StateFlow il collecting dal flusso non attiva alcun codice produttore
- Uno StateFlow è sempre attivo e in memoria e diventa idoneo per la Garbage Collection solo quando non ci sono altri riferimenti ad esso
- Quando un nuovo consumatore inizia a collecting dal flusso, riceve l'ultimo stato nel flusso e tutti gli stati successivi
  - Puoi trovare questo comportamento in altre classi osservabili (observable data holder classes) come LiveData

  (maggiori differenze: https://developer.android.com/kotlin/flow/stateflow-and-sharedflow#livedata)

Usa le coroutine di Kotlin con componenti Lifecycle-aware

- Sappiamo che… Le coroutine di Kotlin forniscono un'API che ti consente di scrivere codice asincrono
- E sappiamo che … Con le coroutine Kotlin, si può definire un CoroutineScope, che aiuta a gestire quando le tue coroutine devono essere eseguite
- Ogni operazione asincrona viene eseguita all'interno di un determinato ambito (scope)
- I componenti Lifecycle-aware forniscono un supporto integrato per le coroutine per gli ambiti logici, insieme a un livello di interoperabilità con oggetti observable, come StateFlow

Approfondimento

- I componenti «sensibili al ciclo di vita» (Lifecycle-aware) eseguono azioni in risposta a un cambiamento nello stato del ciclo di vita di un altro componente, ad esempio activity
  - Questi componenti consentono di produrre codice più organizzato e spesso più leggero, più facile da mantenere

Lifecycle-aware coroutine scopes

- I componenti Lifecycle-aware definiscono i seguenti ambiti predefiniti che puoi usare nella tua app:
  - **ViewModelScope**
  - **LifecycleScope**

ViewModelScope (Ripasso..)

- Viene definito un ViewModelScope per ogni ViewModel nella tua app
- Qualsiasi coroutine avviata in questo ambito viene annullata automaticamente se ViewModel viene cancellato.
- Le coroutine sono utili qui per quando hai un task che deve essere fatto solo se il ViewModel è attivo
  - Ad esempio, se stai elaborando alcuni dati per un layout, dovresti definire l'ambito (scope) del lavoro in ViewModel in modo che se ViewModel viene cancellato, il lavoro viene annullato automaticamente per evitare di consumare risorse
- È possibile accedere a CoroutineScope di un ViewModel tramite la proprietà viewModelScope di ViewModel

LifecycleScope

- Un LifecycleScope è definito per ogni oggetto Lifecycle
- Qualsiasi coroutine avviata in questo ambito viene annullata quando il ciclo di vita viene distrutto
- È possibile accedere a CoroutineScope del ciclo di vita tramite le proprietà lifecycle.coroutineScope o lifecycleOwner.lifecycleScope

Lifecycle-aware flow collection

- Se hai solo bisogno di eseguire una collection che tenga conto del ciclo di vita su un singolo flusso, puoi utilizzare il metodo Flow.flowWithLifecycle()

```
viewLifecycleOwner.lifecycleScope.launch {
    exampleProvider.exampleFlow()
        .flowWithLifecycle(viewLifecycleOwner.lifecycle, Lifecycle.State.STARTED)
        .collect {
            // Process the value.
        }
}
```

Lifecycle-aware flow collection

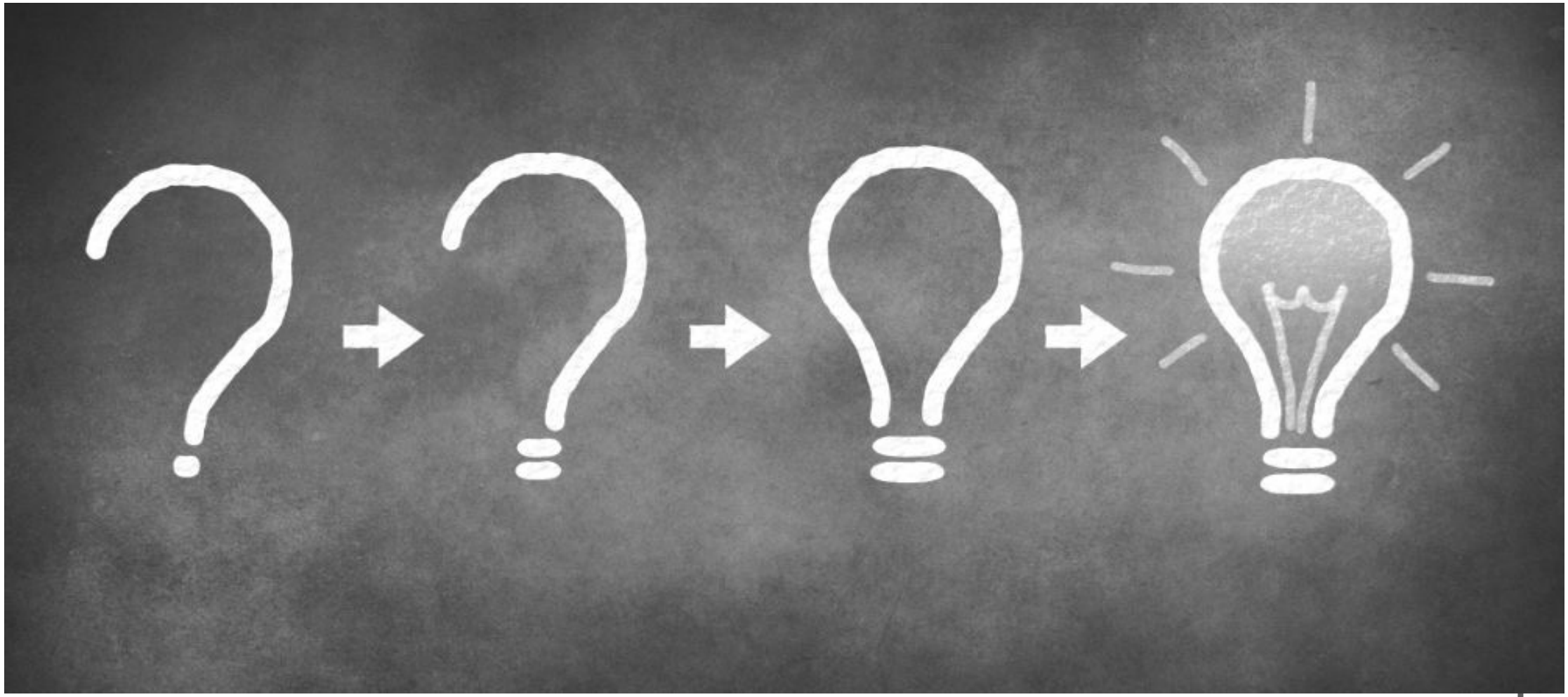
- Tuttavia, se è necessario eseguire la lifecycle-aware collection su più flussi (flow) in parallelo, è necessario raccogliere ogni flusso in diverse coroutine
- In tal caso, è più efficiente utilizzare repeatOnLifecycle() direttamente

Esempio

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
        // Because collect is a suspend function, if you want to
        // collect multiple flows in parallel, you need to do so in
        // different coroutines.
        launch {
            flow1.collect { /* Process the value. */ }
        }

        launch {
            flow2.collect { /* Process the value. */ }
        }
    }
}
```

Domande???



Riferimenti e Approfondimenti

- https://developer.android.com/kotlin/coroutines
- https://medium.com/androiddevelopers/coroutines-first-things-first-e6187bf3bb21
- https://medium.com/androiddevelopers/cancellation-in-coroutines-aa6b90163629
- https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471
- https://kotlinlang.org/docs/coroutines-guide.html
- https://developer.android.com/codelabs/advanced-kotlin-coroutines#7
- https://developer.android.com/kotlin/flow
- https://developer.android.com/topic/libraries/architecture/coroutines
- https://developer.android.com/topic/libraries/architecture/lifecycle