

07 COOROUTINES

Meccanismo per ottenere **programmi concorrenti** senza interagire con thread → forniscono un API vicino alle necessità dei programmatori.

- Capaci di **ricordarci dove ci siamo stoppati** nel caso in cui stoppiamo così poi ripartiamo da dove ci eravamo portati.
- Mantengono lo stack della storia fuori dal vero stack → in un oggetto che ottiene il solo spazio di cui necessita (non extra space). Una struttura friendly del programmatore.
- Molto viene fornito dalle librerie e quindi può essere customizzato facilmente.

Una coroutine può essere definita come un'**istanza di un suspendable computation**:

- That is, a computation that can suspend at a given point, freeze its own state, and later be resumed, possibly in different thread from the one in which it started.
- Il thread X che stava svolgendo c1, dato che c1 è freezata, può fare altro
 - Mentre sto aspettando che un server mi risponda ad una domanda, posso fare una domanda ad un altro server e etc...
- Paradigmi di **programmazione cooperativa**:
 - Esecuzione asincrona e concorrente → varie attività in maniera concorrente: in un giorno studio ma anche mangio
 - Futures
 - Generators
 - Channels
 - Flows
- Dunque, si possono esprimere logiche sequenziali

Con le coroutines è tutto più semplice → **await** per indicare che devo aspettare qualcosa, non ci piacciono le callback perché è un casino (promise).

- Quando una funzione standard è eseguita, svolge fino alla fine o fino ad un punto failure.
- Se qualcosa deve essere eseguito nel mentre, può essere eseguito/schedulato su un thread differente.
- L'idea è di evitare che il OS blocchi il thread → ogni thread deve procedere individualmente senza interferire con gli altri.
- Cooroutines **si basano su thread**, nascondendoli al programmatore → dettagli del thread vengono manipolati e se ne prendono cure le librerie stesse.
 - obiettivo: provare a fare in modo che ci siano più thread oppure sospendere la computazione. → freeze me now e rendi il thread disponibile per schedulare altro
 - *Sospendere la computazione → ho contattato il server, ho mandato i miei messaggi, ora devo aspettare → freezami rendendo il thread disponibile ad altre funzioni → SUSPENDABLE COOROUTINES.*

Per ogni thread possono esistere migliaia di coroutines, solo una alla volta sarà eseguita in un determinato momento dal thread. Quando una coroutine si sospende, thread passa a svolgerne un'altra. Dunque, dall'esterno, si vede che un thread fa più lavori.

Nel caso in cui ci siano molte operazioni bloccanti, si opta per una larga thread-pool. Serve uno scheduler che decida quale eseguire.

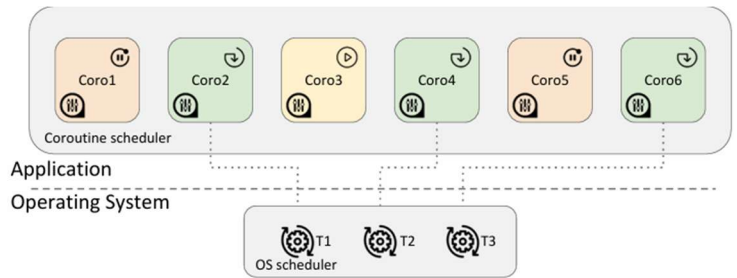
Possiamo avere un numero di task concorrenti maggiore di quanti ne sarebbero supportati dalla cpu.

Per usare le coroutines bisogna importare

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.8.0"
```

La nostra applicazione vede un insieme di **task**:

- **Active state** → correntemente eseguiti da un thread
- **Suspended** → sono in uno stato di pausa
- **Run** → potenzialmente runnable ma non trova un thread per essere eseguita



Quando il quantum si esaurisce, il OS blocca un task e fa in modo che ne parta un altro; Le coroutines vanno avanti ad eseguire finché non trovano un punto di sospensione → Bisogna disegnare le coroutines tenendo conto che una coroutines inizia quando un task raggiungere un punto di sospensione.

Suspend fun

Funzione che può essere eseguita solo nel contesto delle coroutines.

Dopo una sospensione, l'esecuzione di una suspend fun rimane bloccata finché la condizione di sospensione rimane valida. Dunque, **l'esecuzione** di una suspend fun avviene **per sprint**:

- Da inizio Fino a primo suspendable point
- Da primo Fino a secondo suspendable point
- Etc..

I vari sprint possono essere svolti da thread diversi.

Nota: una funzione normale non può chiamare una suspend fun.

- Funzione normale mantiene il proprio stato nello stack
- **Suspend fun mantiene variabile locali** in un oggetto allocato **sullo heap** e se una suspend fun chiama una suspend fun, l'oggetto della nuova s.fun viene linkato a quello che l'ha chiamata e etc.
 - *Si crea quindi una lista e quando una suspend fun viene terminata, si accorcia la LinkedList*

Come posso **startare coroutines** dato che non posso lanciarle da una funzione normale??

→ Android fornisce CoroutineScope che possono essere usati per iniziare nuovi coroutines

Le coroutines possono essere create solo dentro il **CoroutineScope** che comprende/encompas il loro lifetime e offre un insieme di funzioni di costruzione per creare nuove coroutines.

- Ogni scope incapsula un **CoroutineContext** che controlla il comportamento di run-time delle coroutines generato da esso (*spawned from it*).

Durante la vita delle coroutines è possibile alterare il Context → quando introduciamo una suspend fun, quando **invochiamo altre suspend fun** compare un simbolo speciale ➡ (quando si invoca un'altra suspend fun, si crea un punto in cui è possibile sospendere la suspend fun)

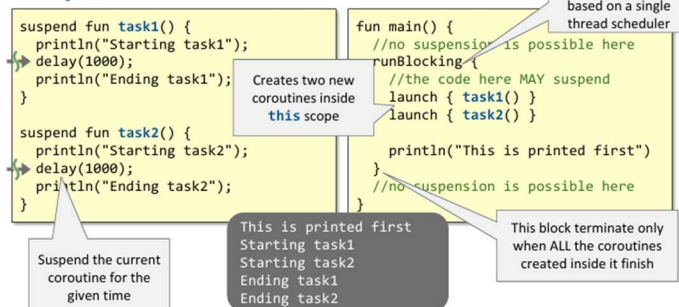
CoroutineScope è allo stesso tempo:

- **interfaccia** → perché fornisce un CoroutineContext
 - esempi:
 - launch
 - async
 - garantisce che, se una coroutine è creata da una funzione di creazione di coroutines, questa coroutine ha lo stesso ciclo vita dello scope (al massimo)
 - runblocking → estende scope a quello delle coroutines
 - le coroutines vengono uccise alla fine dello scope (la vita dello scope non viene intaccata dalle coroutines → se una coroutine finisce, lo scope non finisce con lei)
- **funzione** → crea un nuovo Scope
 - passando un CoroutineContext
 - quando lancia le coroutines usa determinati aspetti
 - specifici coroutines handler
 - specifici tool di debugging
 - molto pericoloso creare un proprio Scope

Metodi forniti nell'istanza CoroutineScope:

- **Launch:** non mi interessa il risultato di una coroutine ma solo che avvenga
- **Async:** voglio che la coroutine mi fornisca un risultato che controllerò
 - Se quando mi servirà il risultato, questo non sarà pronto, mi sospenderò e attenderò il risultato (await)
- **LifecycleScope:** mi garantisce che ogni coroutine creata sarà nel lifecycle della mia attività
 - Se ci sarà una coroutine ancora attiva quando finisce la vita della mia attività → sarà uccisa
- **viewModelScope:** mi garantisce che la coroutine morirà nel caso sia ancora attiva alla morte del viewModel
 - nota: viewModel dura più di lifecycle
- **runBlocking:** lancia una nuova coroutine, bloccando un thread corrente finché essa e tutte le sub-coroutines saranno finite.

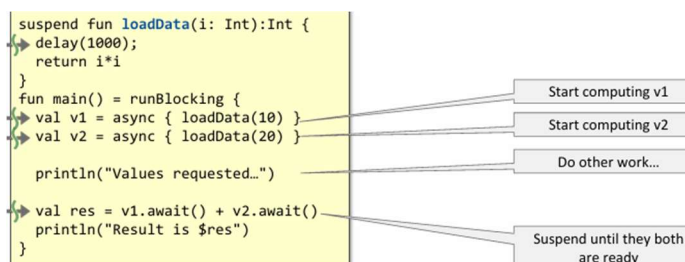
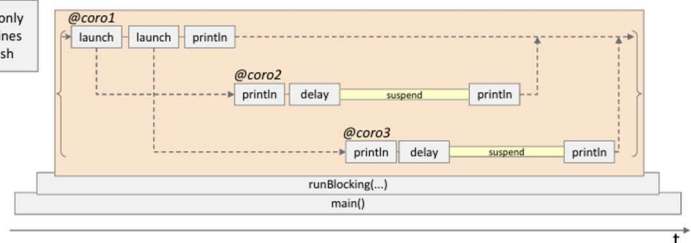
Example



Launch: è conveniente per lanciare cose di cui non mi interessa il risultato al momento.

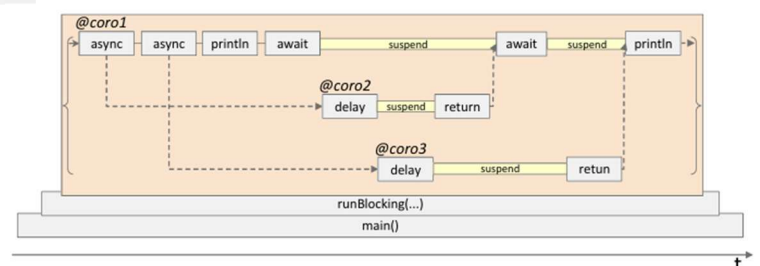
runBlocking all'inizio sa che c'è una singola coroutine da fare. La seconda **launch**, notifica allo scheduler che c'è un altro task da fare e che quindi in caso di sospensioni, ha quel task da eseguire.

Questo avviene perché usiamo **runBlocking** che permette un thread alla volta in esecuzione.



Ci sono situazioni dove serve un risultato per poter continuare le mie cose → uso il metodo **async** che ritorna un oggetto **Deferred<T>**, dove T è il tipo ritornato. Se tutto va bene la computazione ritorna T ma se succede qualcosa → problemi.

Per questo c'è il metodo **await()** → aspetta che ritorni il valore e in caso di errore, invece di ritornare il valore, re-throws l'eccezione.



Apparentemente launch è simile a lanciare un thread.

Le coroutines mantengono le **relazioni padre-figlio** (a differenza di quanto accadeva nel sistema operativo).

Nel caso in cui ci sia un fallimento di una coroutine, il padre ottiene una notification che lo informa che uno dei suoi figli ha avuto un fallimento e quindi anche lui avrà un fallimento.

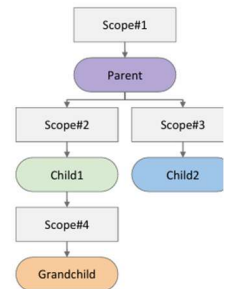
Se il **padre finisce** il suo lavoro, deve aspettare che tutti i figli finiscano too.

Se il **padre è cancellato**, tutti i figli sono cancellati.

Questo è possibile perché `coroutinesScope`, oltre ai metodi per creare le coroutines, mantiene le informazioni su dove vengono create le coroutines, sottoforma di un albero.

Structured concurrency

```
fun main() = runBlocking { //this: CoroutineScope#1
    launch { //this: CoroutineScope#2
        delay(1000)
        launch { //this: CoroutineScope#4
            delay(250)
            println("Grandchild finished")
        }
        println("Child 1 done")
    }
    launch { //this: CoroutineScope#3
        delay(500)
        println("Child 2 done")
    }
    println("Parent done")
}
```



NOTA: kotlin fornisce anche dei **scope già fatti** (ready made):

- **MainScope**: la sua vita è legata al MainThread
 - Disponibile solo quando il main thread è basato su looper
 - Può essere usato quando creiamo una GUI desktop application
- **GlobalScope** incapsula un thread pool contenente tanti thread quanti sono i core disponibili

IN ANDROID

In android abbiamo

- **viewModelScope** che è una proprietà della viewModelClass
 - ereditato da ogni viewModel (che estende viewModelClass)
- **lifecycle scope**: garantisce che resta in vita finchè l'attività resta in vita
- **mainScope (minuscolo)**: scope speciale che esiste finchè esiste la nostra applicazione esiste.
 - Se schedulo coroutines usando il mainScope, sono sicuro che vengano eseguite.

NOTA: in android non usare runBlocking

Coroutine context

Lo scope permette di creare coroutines ma necessita di un coroutines context:

- ha la configurazione del nostro scheduler
- **CoroutinesDispatcher**: definisce l'insieme di thread che possono eseguire le coroutines
- **Job** → rappresenta lo stato della computazione della coroutine corrente
- **CoroutineExceptionHandler**: interviene solo per lanciare coroutines
 - Fixare qualcosa
 - Prevenire il mio fallimento
- **CoroutineName**: stringa utile per debugging

Context sono tipicamente creati aggiungendo gli elementi di cui sono composti:

◦ `val ctx: CoroutineContext = Dispatchers.IO + SupervisorJob()`

Dispatchers: definisce quale/quali thread saranno disponibili per eseguire la computazione suspendable e quale strategia di scheduling useranno.

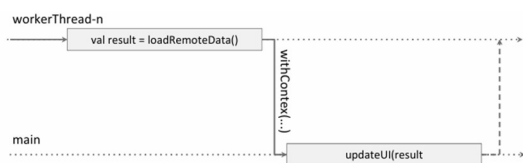
- **.Main**
 - Contiene solo il main thread
 - Disponibile solo in Android e per applicazioni GUI-based
 - Usato per accedere la UI
- **.IO**
 - Basato su una larga thread-pool (~50threads) ottimizzata per il disco e per IO di rete
 - Usata per leggere/Scrivere file, accedere al database e alla rete
- **.Default**
 - Contiene una piccola thread-pool (min2), ottimizzata per lavoro intensivo CPU
 - Usata per task come ordinamento, parsing JSON, cifratura
- **.Unconfined**
 - Usa un qualsiasi thread disponibile, senza restrizioni → NON CI PIACE

Settare un custom dispatcher

- Può essere passato come un argomento della funzione builder
 - L'intera coroutine sarà basata su esso

- Alternativamente è possibile controllare quale specifica parte di coroutine sarà eseguita usando un dato dispatcher usando **withContext()**

```
◦ myScope.launch(Dispatchers.IO) { ... }  
◦ lifecycleScope.launch(Dispatchers.IO) {  
    val result = loadRemoteData()  
    withContext(Dispatchers.Main) { updateUI(result) }  
}
```



```
fun loadImage(imageURL: String) {  
    val job = MainScope().launch {  
        imageView.setImageResource(  
            R.drawable.loading)  
        )  
  
        val bmp = withContext(Dispatcher.IO) {  
            loadRemoteUrl(imageURL)  
        }  
  
        imageView.setImageDrawable(bmp);  
    }  
}
```

MainScope → uso il main thread e preparo un task che sarà eseguito qui. Nel main thread, immediatamente manipolo la mia ImageView settandola come una risorsa che contiene il loading (placeholder che indica che qualcosa sta accadendo). Successivamente uso **dispatcher IO** → carico un url remoto, coroutine è sospesa e viene creata una nuova coroutine con il job di caricare url remoto e ritornare il valore. Main thread è libero, e può fare ciò che vuole, mentre i thread del dispatcher IO stanno lavorando per loadRemoteUrl, successivamente ritornano con il risultato che sarà propagato al thread main che ora può fare ciò che vuole.

Context Handling

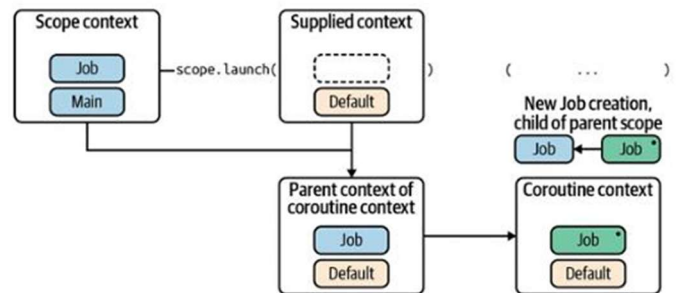
Dentro uno scope, quando creo una nuova coroutine (launch/async) quello che accade è che è creato un **nuovo scope** che incapsula il suo context. (ogni coroutine ha uno scope che è derivato dallo scope del padre con possibilmente qualche modifica nel context).

C'è una parte non copiabile → JOB → ogni coroutine ha il proprio job che è linkato a quello del padre.

Dunque, **JOB** è estremamente importante in quanto definisce lo stato della coroutine come combinazione di 3 valori booleani:

- **isActive**
- **isCompleted**
- **isCanceled**

Nota: job ha un link al job del padre e un link al job del figlio.

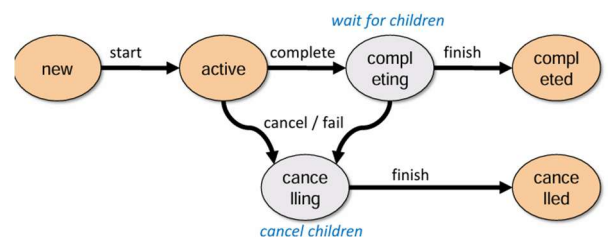


L'evoluzione dello stato non è solo legata a quello che la coroutine sta facendo ma anche a cosa le coroutine a lei legate stanno facendo.

- Tipicamente se **un job è cancellato** → i figli sono cancellati
- Tipicamente se un **job fallisce** → il padre fallisce

Nota: se uno scope contiene un **SupervisorJob** → il failure di una coroutine non causa il failure delle coroutines sorelle ("sibling coroutines").

- **new:** F F F → computazione deve ancora iniziare
- **Active:** T F F → iniziata ma non ha raggiunto la sua fine
- **Completing:** pronta a completare ma sto aspettando figlio (stato temporaneo)
 - Nel caso in cui un mio figlio fallisce, vado in cancelling
- **Cancelling:** se exception/cancellationRequest (stato temporaneo)
 - Propago cancellazione ai miei figli
- **Completed:** T T F
- **Cancelled:** T F T



Jobs: quando invoco launch → un job ritorna e posso fare diverse operazioni:

- Fun Start(): starta una coroutine legato al job
- Fun Cancel (Cause) → cancella il lavoro, passando (opzionalmente) una causa
- Suspend fun join() → sospende l'invocazione della coroutine finché il job è completato

Gestire eccezioni in coroutine:

- La natura fire and forget della funzione **launch** non richiede la gestione esplicita delle eccezioni generate dalle coroutine compilate. Comunque, è possibile installare nel coroutine context un CoroutineExceptionHandler.
- Solitamente le coroutine figlie non si preoccupano di catturare le eccezioni perché ci pensano i genitori.
 - Tranne nel caso in cui sia presente un **SupervisorJob**
- Coroutines create con funzione **async**, catturano sempre le proprie eccezioni e le lanciano quando awayt() viene invocato.

Coroutine cancellation: coroutine possono essere cancellate

Cancellation viene controllata solo quando si raggiunge un punto di sospensione. All'interno della coroutine bisogna inserire **yield** → anche se ho altro lavoro da fare faccio una piccola sospensione da 0ms per verificare se c'è qualcosa da fare.

```
suspend fun decrypt(ins: InputStream, outs: OutputStream, key: Key) {
    val buffer = ByteArray(2048)
    while (true) {
        val size = ins.read(buffer, 0, buffer.size)
        if (size == -1) break
        outs.write( process(buffer, size, key) )
        yield()
    }
}
```

Job definisce come la cancellazione va propagata:

- **Simple Job** → propagata a coroutines sorelle/siblings causando la loro cancellazione
- **SupervisorJob** class → cancellazione non propagata alle sorelle
 - Per installare un supervisor job bisogna usare supervisor scope

Nota: SupervisorScope → crea uno CoroutineScope derivato dall'attuale con supervisorJob dentro esso

- **launch(...)** spawns cancellable tasks (a.k.a. Fire and forget)
 - A coroutine can be created to execute a subtask, without waiting for its completion, somewhat similarly to what can be done spawning a new thread, dedicated to execute the task
 - However, differently from threads, coroutines can be easily cancelled

```
val scope = CoroutineScope(Job())
scope.launch {
    while(true) {
        delay(1000)
        println("Coroutine - tick")
    }
}
//...do other work
scope.cancel()
```

```
val t = Thread {
    try {
        while(!Thread.currentThread()
            .isInterrupted) {
            Thread.sleep(1000)
            println("Thread - tick")
        }
    } catch (e: InterruptedException){}
}.apply { start() }
//...do other work
t.interrupt()
t.join()
```

Se si lancia qualcosa, l'oggetto ritornato può essere usato per cancellare; si può cancellare coroutine anche invocando scopeCancel: tutte le coroutine create saranno cancellate.

Alcune volte si vuole eseguire un job per un tempo limitato, si può usare il costruttore

withTimeoutOrNull che crea un costruttore che:

- Se il figlio sarà completato entro il dato timeout → risultato della coroutine ritornato
- Se non completato → Job cancellato e torna null

CompletableFuture

```
val res = runBlocking {
    withTimeoutOrNull(1500) {
        val d1 = async { task1() }
        val d2 = async { task2() }
        d1.await() + d2.await()
    }
}
```

```
val t1 = CompletableFuture
    .supplyAsync { task1() }
val t2 = CompletableFuture
    .supplyAsync { task2() }
val all = CompletableFuture
    .allOf(t1, t2)
    .orTimeout(1500,
        TimeUnit.MILLISECONDS)

val res = try {
    all.join()
    t1.get() + t2.get()
} catch (e: CompletionException) {
    null
}
```

Nota: con i thread sarebbe molto più complesso, bisognerebbe fare supply e poi waitOrTimeout → molto complesso

Sincronizzazione

L'esecuzione concorrente può causare interferenze → per il modo in cui le coroutine sono disegnate ci sono molti luoghi in cui si rischia interferenza ma possono essere **semplificati**.

Tipicamente le interferenze sono causate dall'accesso alla stessa variabile da diversi thread. Per questo motivo kotlin fornisce librerie che forniscono **2 classi** che supportano sincronizzazione:

- **Mutex**: quando provo a fare lock, non blocco il thread ma il **thread può fare altre cose** finchè la risorsa è disponibile
- **Semaphore**: cnt deve essere > 0, quando prendo devo decrementare il semaforo.

Ci sono altri modi per parallelismo:

- Si può usare un context `limitedParallelism(1)`: fa in modo che ci sia un unico thread a fare quella determinata cosa.
 - *Se la nostra sezione critica non ha critical point, la nostra sezione sarà eseguita da un solo thread. Mentre se ci sono dei punti di sospensione, posso andare su altri thread o anche se resto sullo stesso thread, rischio che una funzione modifichi le mie cose → devo gestire.*

Coroutines mutex

Mutex ha due stati:

- Lock
- Unlock

Si può usare la extension function **withLock** che causa la sospensione della coroutine finchè il lock non sarà acquisito:

- Quando lock sarà acquisito → lavora
- Finisce quello che deve fare e lo rilascia.

```
fun main() = runBlocking<Unit> {
    var mutex = Mutex()
    var n = 0;
    runBlocking(Dispatchers.Default) {
        launch {
            for ( i in 1..1_000_000 )
                mutex.withLock { n++ }
        }
        launch {
            for ( i in 1..1_000_000 )
                mutex.withLock { n++ }
        }
    }
    println("Final value: $n")
}
```

Questi due task potrebbero essere eseguiti in parallelo e quindi otterremmo un valore minore a 2milioni; avendo una variabile condivisa con lock, il problema non si pone e sarà stampato 2mln.

Coroutines semaphore

Necessita di essere creato con un nr di permessi iniziale.

withPermit{} → quando chiamato decremento di 1, quando finito incrementa di 1.

Alcune **regole**:

- Cnt non può essere minore di 0, quindi se si vuole ottenere un permesso e cnt==0 → bisogna aspettare
- Cnt non può mai essere superiore al valore iniziale, quindi se si vuole rilasciare e cnt==val_in → bisogna aspettare

```
suspend fun loadData(name: String) {
    println("Begin $name")
    delay((Math.random()*300+500).toLong())
    println("End $name")
}

fun main() = runBlocking<Unit> {
    var semaphore = Semaphore(2)
    runBlocking(Dispatchers.Default) {
        for (i in 0..3)
            launch {
                semaphore.withPermit {
                    loadData("t$i")
                }
            }
    }
}
```

Begin t0
Begin t1
End t1
Begin t3
End t0
Begin t2
End t2
End t3

Nota: un semaforo con un solo permesso è simile ad un Mutex.

Pattern di uso tipici di Android

- activity ha `lifecycleScope` che permette di creare coroutine che durano tanto quanto l'activity
- viewModel ha `viewModelScope`
- Composable ha **rememberCoroutinesScope()** ritorna un `coroutineScope` che dura al massimo quanto il composable presente sullo schermo

--esempio su kotlin → vedi lezione r15 ultimi 10 minuti

Immaginando di avere una userInterface che permette di inserire credenziali → può essere conveniente avere un bottone vicino al campo password che permette di mostrare la pw per pochi secondi.

```
@Composable
fun SomeWidget(...) {
    val reveal = viewModel.isShowingPassword.observeAsState()
    val scope = rememberCoroutineScope()
    Row {
        if (reveal?.value == true) { Text(viewModel.password) }
        else { Text("****") }
        Button(onClick = { scope.launch { viewModel.showPasswordBriefly() } }) {
            Text("Reveal password")
        }
    }
}

class SomeViewModel: ViewModel {
    val isShowingPassword = MutableLiveData<Boolean>(false)
    suspend fun showPasswordBriefly() {
        isShowingPassword.value = true
        delay(3000)
        isShowingPassword.value = false
    }
    //...
}
```

ViewModel che contiene un mutable state booleano `isShowingPassword`. Nel Composable ho il widget che è responsabile per mostrare la password. Controlla dal ViewModel il campo `isShowingPassword`. Nel viewModel c'è una suspend function che setta `isShowingpw` a true, delay per un po' di tempo e ritorna a false. Il mio Composable ha una Row che si comporta diversamente in base al `showingPw`. Coroutine `scope.launch` → invoca il metodo `suspendable`.

Un altro pattern possibile di android serve a fetchare dati da una sorgente di dati remota:

```
class MyRepository {
    suspend fun getMyData(id: Int): Result<MyData> =
        withContext(Dispatchers.IO) {
            val localData = async { myDAO.getMyData(id) }
            val remoteData = async { myAPI.getMyData(id) }
            try {
                val l = localData.await()
                val r = remoteData.await()
                if (r.isValid()) return Result(r,null)
                else return Result(l,null)
            } catch (error: Throwable) {
                return Result(null,error)
            }
        }
}
```

Repository: oggetto che può fetchare informazioni localmente e da remoto.

Un server remoto non è sempre contattabile, infatti è comune avere una soluzione alternativa: invece di mostrare un errore all'utente posso mostrare dei dati che avevo mantenuto in cache locale. Vorrei prendere queste informazioni da entrambe le parti (remoto e locale). Se in un arco decente di tempo riesco ad ottenere da remoto, bene; se dopo un determinato tempo non riesco, uso informazioni locali.

La funzione ritorna un result che può essere successful o errore.

Per invocare `getMyData`, il viewModel deve avere una suspend fun capace di fare questo.

Connessioni di rete non devono mai andare su `threadMain` infatti `getMyData` usa `withContext` che è una suspend fun responsabile di eseguire coroutine su un contesto diverso, in questo caso `Dispatches.IO`.

`LocalData` e `remoteData` create in due `async` coroutine figlie → se `remoteData` mi fornisce un risultato valido, ritorno dati remoti.

Se qualcosa non va, ritorno `localData` o errore nel caso in cui ci siano errori.

Se ci sono altri problemi tornerà errore.

```
suspend fun deleteAllNotes() = withContext(Dispatchers.IO) {
    // This creates a new scope and suspends the current coroutine
    // until all child coroutines in it have completed

    // Equivalent to: do these things in parallel then return
    // when they are all done
    coroutineScope {
        launch { remoteDataSource.deleteAllNotes() }
        launch { localDataSource.deleteAllNotes() }
    }
}
```

Creo uno scope temporaneo che mi permette di lanciare altro.

Interessante perché in alcuni casi ho bisogno di cancellare entrambe le operazioni e avendo le due coroutine lanciate da un `ChildrenScope`, posso semplicemente cancellare il `childrenScope`.

Invocare repository:

Le repository sono spesso accedute da un viewModel, questo è il motivo per cui il viewModel ha il proprio `viewModelScope` che garantisce alle coroutine di essere vive finché il viewModel è vivo.

Il `coroutineScope` contiene un **SupervisorJob**, dunque se ci saranno molte operazioni, il fallimento di una operazione non influenzerà le altre. (solitamente il viewModel fornisce molte cose a cose indipendenti tra loro).

Inoltre, viene usato il `Dispatchers.Main` dunque tutte le invocazioni/creazioni di coroutine fatte dal `viewModelScope` iniziano con il thread main.

Anche le attività hanno il loro scope → `lifecycleScope` che causa la cancellazione di tutte le coroutine nel momento in cui l'attività finisce.

```
class MyViewModel(val repo: MyRepository): ViewModel() {
    fun getMyData() {
        viewModelScope.launch {
            val data = repo.getMyData()
            if (data.error != null)
                //display error
            else
                //update UI state
        }
    }
}
```

Creo una coroutine che parte dal thread main e provo ad invocare la repository per ottenere informazioni e se non riesco, mostro errore.

Alcune note su coroutines:

- Le JVM non forniscono supporto per le coroutines
- Kotlin implementa coroutines nel compilatore trasformando le funzioni sospendibili in state machines
- Kotlin usa una singola keyword per l'implementazione, il resto è fatto dalle librerie
- Kotlin usa CPS (Continuation Passing Style) per invocare coroutines in quanto non è possibile invocare funzioni sospendibili in altro modo.
- Usando i dispatchers, è facile cambiare thread e mantenere il codice facile da leggere e mantenere

COROUTINES UNDER THE HOOD

Vedere all'interno → cosa succede veramente quando scriviamo `suspend`. Compiler svolge molto lavoro, necessario perché l'esecuzione delle nostre `suspend fun` necessita di andare il più lontano possibile (non necessita di andare alla fine) finché un suspension point è trovato, a questo punto il thread può svolgere un task differente.

Il modo in cui le orchestration appends dipendono dal tipo di thread invocato:

- Looper thread (come il main)
 - Messaggi postati nella **message queue**
 - Si estrae dalla + message queue
 - Dunque, il messaggio di fermare la gestione di questa coroutine e passare a gestirne un'altra va inserito all'interno di questa queue.
- Thread pool
 - Non c'è una message queue
 - Thread mantiene un'altra struttura → **JOB queue**
 - Collezione di runnable da essere eseguiti
 - Synchronization statement (semaphone, variable, etc) che governa l'accesso alla coda con il thread che fetcha il task al top e inizia ad eseguire. Quando il task ritorna, il thread va al prossimo e prova a fetcharlo.
 - Task presi in un dato ordine, possono completarsi in un altro ordine.
- Thread "base"
 - Sarebbe capace di lanciare una coroutine ma quando questa raggiunge un suspension point, il thread si ferma → la coroutine si ferma e non sarà mai finita. Fortunatamente il modo in cui le funzioni sono diseguate nella libreria, previene questa situazione → THREAD "base" non possono lanciare le coroutines.

Stack: contenitore per lo stato, quando una funzione è invocata tutti i parametri e variabili locali sono salvate sullo stack → se il thread viene bloccato dal sistema operativo (o frozen), lo stack rimane come sta e quando il thread riprende a funzionare può andare nello stack e prendere i suoi valori pre-sospensione.

Un modo per sospendere può essere:

- Ho il mio thread con i miei suspension point e quando ne raggiungo uno, io deposito lo stack.
 - Troppo pesante
- Il design delle coroutine non può quindi basarsi sullo stack; infatti, il compilatore deve trasformare cosa il codice contiene in qualcosa'altro → CONTINUATION: codice speciale che viene aggiunto alle `suspend fun`, viene salvato il computation state della fun. Oltre allo state, dentro il continuation ci sono tutte le variabili locali e un link al primo continuation (invoking function). Dunque, quando raggiungo la fine della funzione, so cosa invocare.

Continuation<T>

Interfaccia implementata dalla classe anonima invocata dal compilatore per ogni suspend fun.

Contiene:

- Stato dell'esecuzione della funzione
- Link al primo continuation → funzione che l'ha invocata

Volendo, posso fornire alla funzione, il codice che deve eseguire quando finisce → fornisco un parametro chiamato resumeWith: fornisce un indirizzo che contiene il dettaglio di cosa deve esser fatto dopo. Value T è il risultato che la funzione ritornerà.

Nella continuation posso anche salvare tutte le variabili locali che mi servono.

```
public interface Continuation<in T> {  
  
    /* Resumes the execution of the suspending  
    function passing [value] as return value  
    of the last suspension point */  
    public fun resumeWith(value: T)  
  
    /* Context of the suspending function,  
    and information about the thread  
    to be used when resumed */  
    public val context: CoroutineContext  
}
```

Inoltre, contiene context → per capire quale thread può riesumarmi quando sarò sospesa.

- il codice nella funzione ha l'opportunità di controllare quando e dove la continuation è invocata
 - sincrona → su stesso thread
 - asincrona → su altro thread

Una suspend fun può invocare:

- plain function
 - usa stack
 - non possono esserci sospensioni
- suspend function
 - stack non usato
 - nuova suspend function viene linkata alla continuation corrente

Nota: una regular function non può invocare una suspend function (non ha continuation)

Dunque, il compilatore traduce il corpo di una suspend function in una macchina a stati finiti

- sono presenti N punti di sospensione, FSM ha N+1 stati (inizio + N punti)

La continuation è dunque una closure contenente:

- stato corrente della FSM
- insieme delle variabili locali dentro la suspend function
- risultato della precedente computazione
- callback che sarà invocato quando la funzione termina
- coroutine context con tutti i suoi dettagli
 - Dispatcher → responsabile di invocare continuation
 - Job → contiene informazione su come la coroutine corrente è legata alle altre
 - ExceptionHandler → fallimenti

Continuation passing style

```
object : Continuation<Unit> {  
    private var label = 0 // where to start the execution  
    private var d: SomeData? = null // variable to be preserved when suspending  
    val context: CoroutineContext  
  
    fun resumeWith(result: Any) {  
        when (label) {  
            0 -> { // the function has just been invoked  
                label = 1  
                var tmp1 = fetchData(this)  
                if (tmp1 == COROUTINE_SUSPENDED) return  
                resumeWith(tmp1)  
            }  
            1 -> { // fetchData() has terminated successfully  
                d = result as SomeData  
                label = 2  
                var tmp2 = processData(d, this)  
                if (tmp2 == COROUTINE_SUSPENDED) return  
                resumeWith(tmp2)  
            }  
            2 -> displayData(result as SomeData)  
            else -> throw IllegalStateException()  
        }  
    }  
}
```

```
suspend fun showData() {  
    val d = fetchData()  
    val r = processData(d)  
    displayData(r)  
}
```

il compilatore trasforma il giallo in arancione

label: variabili che mi dice dove sono all'inizio
d: risultato della prima cosa, non viene consumato → deve resistere a sospensione
r: sarà eliminato → viene creato dopo il secondo suspension point ed è consumato interamente in displayData → non è una variabile che deve esistere anche oltre sospensione
context: coroutine context
implementation della interfaccia: fornisce il metodo resumeWith che ha Any come parametro → cosa sarà passato.

Se sono 0, sarò 1; se sono 1, sarò 2; se sono 2, mostro dati

PSEUDO_CODE

Nota: il fatto che invochiamo una suspend fun, non implica che sarà sospesa, possono anche non accadere sospensioni.

Due tipi di funzioni che sospendono sempre:

- Delay
- WithContext

- delay(milliseconds: Int) – causes the current thread to be suspended and resumed in the given number of milliseconds
- withContext(ctx: CoroutineContext) { λ } – suspends the current thread, delegating the execution to any thread inside the given context; current thread will be resumed with the value/exception returned by the provided lambda

FLOWS

Quando vedi il flusso, tu vedi il valore corrente. Se c'è un cambiamento, prendi il nuovo valore. Vedi il valore che ha ora la variabile e quale avrà → puoi vedere l'evoluzione della variabile. Tipicamente una coroutine ritorna un valore → può essere una collection.

Nota: ci sono alcuni casi in cui io non voglio aspettare, magari ho la prima riga già disponibile da tempo.

Può essere comodo, avere una coroutine che non ritorna un single value alla fine ma un flusso di valori → coroutine ritorna immediatamente un flusso che tu puoi osservare. Coroutine inserisce dati in questo flow e tu puoi tirarli fuori in diretta.

→ Ti permette di reagire all'evoluzione delle cose in time

Come abbiamo qualcosa lo schermo viene popolato e mostrato il tutto.

Per questo kotlin introduce il tipo **Flow<T>** che rappresenta un data stream asincrono che emette valori in modo sequenziale. Un Flow trasporta un numero di elementi sconosciuto.

- Puoi ottenere un elemento alla volta con la possibilità di ispezionare, manipolare, etc

È **rifornito** da un sorgente che inserisce cose nel flow. → continua a produrre finché gli item sono finiti oppure fallisce.

- Se fallisce deve indicare il failure e non deve provare ad andare avanti → flow is over.

È **consumato** da un consumer che prende i dati.

È un'interfaccia con un insieme di **extensions function**: map, filter, take, zip.

- Come si ottiene qualcosa, subito applicabili
- Operano su un upstream flow e producono un downstream flow
- Quando invocati, svolgono una serie di operazioni per esecuzione futura e ritornano velocemente

Flow opera lazy → non si vede nulla finché non c'è un consumer che richiede dati → quando dati richiesti, vengono emessi.

Presenta degli **operatori terminali** (collect, single, reduce, toList) o operatori che triggerano l'esecuzione dei passi futuri del processo

- Completano normalmente o con eccezione

Flow sono eseguiti in maniera **sequenziale** all'interno della stessa coroutine

- Poche operazioni sono designate a introdurre concorrenza nell'esecuzione del flow

→ se i dati che cerchiamo non ci sono, mi fermo ad aspettarli.

Ci sono delle operazioni che possono essere scartate in base al momento in cui lo ricevo.

Ci sono due tipi di flow:

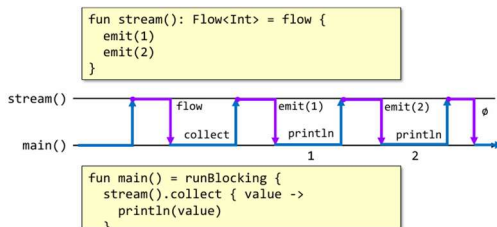
- **Cold flow** → completamente lazy
 - Se due consumer diversi si collegano ad un cold flow, cold flow sarà eseguito due volte
 - Prima volta per un consumer
 - Seconda volta per un consume
- **Hot flow**
 - Capace di emettere valori indipendentemente dalla richiesta di un consumer
 - Capace di mandare in broadcast i valori
 - Utile ad esempio per preservare i dati da verticale ad orizzontale
 - Ci dà la possibilità di fornire informazioni a molti

Definizione di Flow:

```
interface Flow<out T> {  
    //Internal method not to be used directly  
    suspend fun collect(collector: FlowCollector<T>)  
}  
  
interface FlowCollector<in T> {  
    //Not thread safe - MUST BE INVOKED SEQUENTIALLY  
    suspend fun emit(value: T)  
}
```

flowConnector: responsabile di emettere valori

Esempio uso:



chiedo per primo, ottengo primo; chiedo per secondo, ottengo secondo.

Ogni volta che qualcosa emit, faccio; se nulla → sospendo e thread libero.

Creazione

- o fun <T> **flowOf**(vararg elements: T): Flow<T>
- o fun <T> Iterator<T>.asFlow(): Flow<T>
- o fun <T> Iterable<T>.asFlow(): Flow<T>
- o fun <T> Array<T>.asFlow(): Flow<T>
- o fun <T> Sequence<T>.asFlow(): Flow<T>
- o fun <T> IntRange.asFlow(): Flow<T>
- o fun <T> (() -> T).asFlow(): Flow<T>
- o fun <T> **emptyFlow**(): Flow<T>
- o fun <T> **flow**(
 block: suspend FlowCollector<T>().->Unit
): Flow<T>

Builder

```
val flowA = flowOf( 1, 2, 3 )

val flowB = { -> listOf("a","b","c") }.asFlow()

val flowC = arrayOf("Hello","World").asFlow()

val flowD = flow { //Lambda function with receiver
    this.emit("Zero")
    for (i in 1..100) {
        delay(100)
        emit("s$i")
    }
    emit("Last")
}
```

flowD: emette stringa Zero e con un delay di 0.1s emette s:: e infine emette Last

Operatori intermedi:

i dati ricevuti dal producer, potrebbero essere non quelli desiderati, usiamo operatori:

- **map(...)** / **mapNotNull(...)**
 - o Transform **Flow<T>** into **Flow<R>**
- **mapLatest(...)**
 - o When the upstream flow emits a new value, computation of the previous transform is cancelled
- **debounce(...)** / **sample(...)**
 - o Drop elements emitted too frequently
- **filter(...)** / **filterIsInstance()**, **filterNot(...)**, **filterNotNull(...)**
 - o Selectively emit elements according to the given predicate
- **flatMapConcat(...)** / **flatMapLatest(...)** / **flatMapMerge(...)**
 - o Apply a transformation that produces a new flow per each element and flatten it using different strategies
- **onStart(...)** / **onEach(...)** / **onEmpty(...)** / **onCompletion(...)**
 - o Perform a given action when the corresponding event happens
- **zip(...)**
 - o Zip value of the current flow with another one, applying a transformation
- Several other operators...
 - o Check the online documentation

debounce: Se ricevo troppo velocemente, scarto

nei flatMap cambia come i vari dati interagiscono tra loro

onStart: lambda da invocare all'inizio

onEmpty: lambda se finisce vuoto

zip: prende un elemento da primo e uno da secondo e crea un pair

combine: invece di forzare di consumare insieme, lui prende un elemento proveniente da un lato e ultimo elemento proveniente dal altro lato e crea pair

- **collect(...)**
 - o Terminal flow operator that collects the given flow with a provided action
 - o If any exception occurs during collect or in the provided flow, it is rethrown from this method
- **single()** / **first()**
 - o Await for the only/first element of the flow
- **toList()** / **toSet()** / **toCollection()**
 - o Collect the flow elements into a destination
- **fold(...)** / **reduce(...)**
 - o Accumulates values in the flow

fold: combinare

Flow vs sequenze

Il flow è powered dalla coroutine mentre sequenze operano in maniera sincrona

- Operations in flows can be suspended, while sequences can only block
 - o This allows the flow to be evaluated concurrently with other tasks, if available

```
fun simple() = flow {
    (1..3).forEach { delay(100); emit(it) }
}

fun main() = runBlocking<Unit> {
    launch {
        for (k in 1..3) {
            println("Coroutine $k")
            delay(100)
        }
    }
    simple().collect { v ->
        println("Flow $v")
    }
}
```

Coroutine 1
Flow 1
Coroutine 2
Flow 2
Coroutine 3
Flow 3

Flow possono essere cancellati mentre sequenze no

```
fun infinite() = flow<Int> {
    var i = 0;
    while(true) { delay(100); emit(++i) }
}

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(350) {
        infinite().collect {
            v -> println(v)
        }
    }
    println("done")
}
```

1
2
3
done

Operatore transform

```
fun main() = runBlocking<Unit> {
    (1..3)
        .asFlow()
        .map { it*2 }
        .transform {
            emit(it-1)
            emit(it)
        }
        .collect { println(it) }
    println("done")
}
```

1
2
3
4
5
6
done

Per ogni elemento in ingresso, ne voglio emettere 2

Conflate → eliminare lenti o veloci

```
fun simple() = flow {
    for (i in 1..3) { delay(100); emit(i) }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().conflate().collect {
            delay(300)
            println(it)
        }
    }
    println("Collected in $time ms")
}
```


1
3
Collected in 738 ms

Scarta cose che arrivano mentre sto caricando

Nota: flow constraints: implementazione dell'interfaccia flow deve

- Context preservation → non possiamo modificare il context dell'esecuzione

- non possiamo cambiare coroutine
- flowOn: permette di cambiare, in un punto della nostra pipeline, i Dispatcher usati downstream per il flow upstream
- channelFlow: incapsula tutto il lavoro del context preservation, permettendo di fare collection e emissioni in diverse coroutine

```
fun main() = runBlocking {  
    flow {  
        emit("Emitted in ${  
            Thread.currentThread().name  
        }")  
    }  
    .flowOn(Dispatchers.IO)   
    .map { it -> "$it\nProcessed in ${  
        Thread.currentThread().name  
    }"}  
    .collect {  
        println(it)  
    }  
}
```

Emitted in DefaultDispatcher-worker-1
Processed in main

trasparenti alle eccezioni: se avviene un'eccezione questa viene propagata giù.

- Una eccezione downstream viene sempre propagata al collector
- Nessuno valore può essere emesso da un blocco try/catch
- Flow can use the catch(...) operator to handle upstream exceptions
 - Terminal operators throw any unhandled exception that occurs in their code or in upstream flows
 - The onComplete(...) operator replaces the finally block

```
flow {  
    emit(1); emit(2); emit(3)  
}  
.map {  
    computeOne(it)  
}  
.catch { ... } // catches exceptions  
                // in emitting data and computeOne  
.map {  
    computeTwo(it)  
}  
.collect {  
    process(it)  
} // throws exceptions from process and computeTwo
```

Se compute 2 ha eccezione, la porta su

JAVARX Observables possono essere trasformati in flows con .asFlow

Flows can be transformed into **Observable** via the **.asObservable()** extension function

```
fun aFlow():  
    Flow<String> {  
    return Mono  
        .just("a")  
        .asFlow()  
    }
```

```
fun anObservable():  
    Observable<String> {  
    return flow {  
        emit("a")  
    }  
        .asObservable()
```