ANDROID THREADING

Ci sono alcune situazioni in cui avere un solo flusso di processo potrebbe essere problematico:

- *Es: faccio richiesta al server, mentre attendo risposta non potrei fare altro se ne avessi solo uno.*

Per risolvere questa cosa → Astrazione thread

Astrazione **thread** → flow computazionalmente indipendente che è schedulato dal OS e opera dentro i processi.
Posso avere più thread e quindi schedulare più flow.

- I vari core della cpu sono condivisi tra i vari thread.

Quando un **processo inizia**, un thread singolo opera al suo interno, il processo **può lanciare altri thread** che condivideranno lo spazio di indirizzamento e saranno capaci di cooperare.

Nota: essere capace di svolgere molte operazioni allo stesso tempo può essere conveniente ma anche una causa di diversi errori.

I thread sono istanza della classe **java.lang.Thread**

- Di default, in java, il modo in cui creiamo i Thread è istanziare la classe Thread che richiede parametri per specificare quale computazione si vuole → oggetto java.lang.Runnable
- Thread devono essere startati esplicitamente per essere schedulati dal S.O.

```
val t = Thread{ println("This is a thread") }.also { it.start() }
```

Quando un **thread è creato**, svolge quello che deve fare finché non termina.
Se un thread (es: main) vuole sapere se suo figlio ha finito, può usare il metodo join; il **metodo join** non è conveniente in quanto è bloccante → attente la terminazione di un thread.

Ogni thread ha **un proprio stack** che deve essere allocato (per salvare variabili locali e indirizzi di ritorno).
Tutti gli oggetti in Kotlin sono allocati su Heap.
Quando due thread vogliono operare su una variabile condivisa, bisogna optare per una **sincronizzazione** altrimenti si rischia di leggere valori non validi.

- In kotlin la classe Any è stata privata del suo behaviour (in java ogni oggetto poteva operare con primitive di sincronizzazione offrendo sia mutex che condition)
- In base al contesto, vengono fornite **varie funzionalità**→ several synchronization primitives may be adopted, both from the standard Java library (java.util.concurrent.*) and from the Android one (android.os.*).

Quando si **crea un'applicazione**, il OS crea un processo e al suo interno il **thread main**

- Costruisce una **coda** che contiene messaggi, eventi inseriti dall'OS derivanti da user (che clicca qualcosa), da OS (che vuole dire che c'è una connessione possibile).
  - La coda può contenere 0+messaggi
- **Loop** infinito:
  - Estrae messaggio dalla coda
  - Esegue
    - Inserisce altri messaggi se bisogna fare altri
  - Continua finché c'è qualcosa in coda.
- Nota: Molti altri thread sono generati automaticamente a partire dal main in modo tale da supportare l'esecuzione della VM e dell'OS
  - *Se dopo un onClick, si esegue un lavoro pesante → applicazione si potrebbe bloccare → bisogna usare thread.*

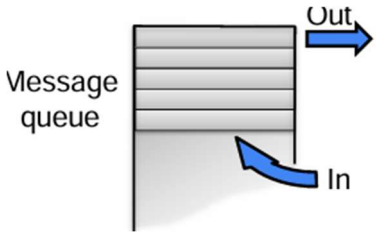
Thread esistenti nella JVM

- **GC**: thread garbage collector a bassa priorità
- **JDWP**: supporta debugginig a ispezione memoria
- **Compiler**: si occupa di compilazione e trasformazione in eseguibile
- **ReferenceQueueD, FinalizerDeamon, FinalizerWatchd**→ finalizzazione oggetti e per ripulire

Android System Threads:

- **Binder_X threads** → gestire intenti e altre richieste IPC
- **Singal catcher** → detecting e gestione segnali provenienti da utenti
- **GL updated**: interagisce con GPU
    - Pushare nuovi dati in GPU
        - Gestisci l'interfaccia utente con accelerazione hardware
- **hwUITask**→ gestire messaggi provenienti da UI
    - button on click

MAIN THREAD



Gestisce coda contenente messaggi → implementa il **loop**

- prende da testa e in base al messaggio invoca funzioni apposite
- messaggi nuovi inseriti in coda

Funzioni:

- **istanzia** components come conseguenza di una richiesta
    - Application e Activity
- **Notifica** agli oggetti (Application e Activity) gli **eventi** legati al loro ciclo di vita
- Invia richieste di **disegno** alle views
- Invia gli eventi legati all'iterazione utente ai corrispondenti listeners
- invoca life cycle metodi

Accesso alla coda è sincronizzato → protetto con mutex. Lock finché non finisce l'attività che sta svolgendo

- Ogni task eseguito sul thread main deve terminare in poco tempo altrimenti si blocca e applicazione non risponde.
  Come svolgere *operazioni lunghe*?
    - Creo un *nuovo thread* dedicato a quelle applicazioni
    - Posso usare *Co-Routines*
        - Alta concorrenza, usando il thread ma non gestendolo

Astrazioni:
Android fornisce diverse astrazioni per supportare il programmatore a sviluppare programmi concorrenti:

- **Looper**:
    - Nota: main thread è un looper → gira di continuo
        - Usufruisce di Handler che dispatcha
- **Handler**: dispatcha le azioni

Looper e Handler vivono insieme perché forniscono lo stesso pattern che è usato dal main thread.

Creare thread:

Sappiamo quando li startiamo ma non siamo a conoscenza di quando vengono uccisi (non ritornano nulla).

```
val t= Thread {
    //the code here will be executed
    //in a secondary thread
  }
t.start();

//… other operations

try {
  t.join(); // wait for thread termination
} catch (ie: InterruptedException ) {
  Thread.currentThread().interrupt();
}
```

Quando killi un processo intero → ok; quando **killi** solo un **thread** → può essere che stava facendo qualcosa e quindi ho una struttura dati inconsistente. **L'utente non può killare un thread** ma si può chiedere di stoppare → a questo punto il thread finisce le cose che doveva fare in modo tale da lasciare tutto consistente e poi si ferma.

Nota: i thread non ritornano nessun valore → si può usare una **variabile share** a cui accedere in modo sincrono con un lock → quando si vuole checkare la variabile, bisognerebbe lockare e poi unlockare dopo il check. *(solo un thread alla volta può possedere il lock)*

```
class SharedObject {
  private var sharedValue=0

  fun increment(): Int {
    return synchronized(this) { ++sharedValue }
  }
  fun decrement(): Int {
    return synchronized(this) { --sharedValue }
  }
  fun getValue(): Int {
    return synchronized(this) { sharedValue }
  }
}
```

***NOTA**: usare priorità negative*

```
fun thread(
  start: Boolean = true,
  isDaemon: Boolean = false,
  contextClassLoader: ClassLoader? = null,
  name: String? = null,
  priority: Int = -1,
  block: () -> Unit
): Thread
```

Problemi relativi ai thread:

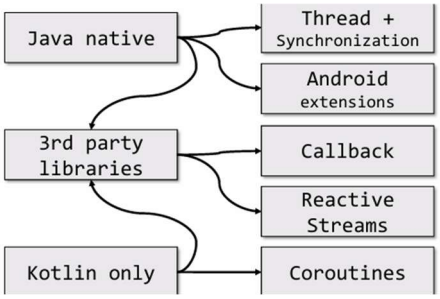
- Tutti i thread accedono allo **stesso spazio di indirizzamento**
  - C'è solo un singolo heap in cui tutti gli oggetti sono allocati.
- Se un riferimento ad un oggetto è conosciuto da 2+ thread → oggetto è condiviso
- Ci sono **interferenze** e risultati imprevisti, se non viene eseguita alcuna sincronizzazione
- Unico modo per comunicare: **shared state**/variable con lock
  - Classe che ha il lock come private value
  - Variable è un private value e ha visibili:
    - Getter → prima di prendere il valore deve acquisire il lock
    - Setter → prima di modificare il valore deve acquisire il lock
- Le attività hanno un **ciclo di vita** → solitamente creiamo thread e vogliamo ottenere risultati.
  - I thread secondari, che creiamo, non sono a conoscenza di questi cicli di vita
    - *Nel caso in cui l'attività è finita e io sto provando a passare i miei dati alla GUI →crash*
    - Per prevenire crash, i thread secondari non possono accedere all'interfaccia grafica, **l'interfaccia grafica può essere manipolata solo dal main thread**.
  - Il main thread non può fare attività lunghe (es: lettura da db, conessioni alla rete, lettura da filesystem)

  → Usare il main thread in modo tale che collezioni le informazioni e passi la gestione ad un altro thread che calcola i risultati e poi ripassi la gestione al main thread che deve controllare se ancora utile
    - Si → procedi
    - No → Drop it

- A part of the Android framework is ONLY accessible from the main thread
  - Typically, the View hierarchy
  - Any attempt to perform operations entailing this kind of objects from a secondary thread will give rise to an exception, terminating the current process

- molto facile creare ma molto difficile cancellare un thread
  - interrupted flag settato attraverso interrupted method

    Interrupt method → setta il flag a true
    - se il thread è in uno stato di wait o sleep, il flag non è settato



JavaNAtive: crei thread usando il costruttore di classe thread o la thread builder function.

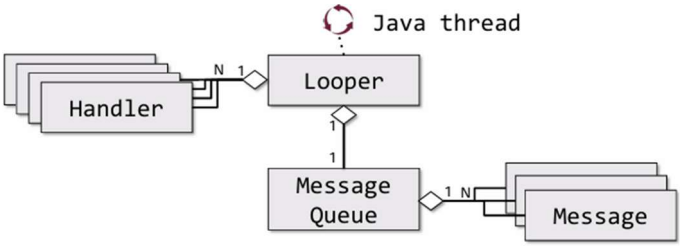
Si possono usare le funzioni Android, le funzioni fornite da **Looper/Handler/MEssageQueue**.

Si possono usare librerie di terze parti (reactive kotlin e reactive java)

Si possono usare le **coroutines** → kotlin only.

NOTA: Inizialmente android forniva un modo per svolgere cose asincronamente → deprecato (AsyncTask)

LOOPER:

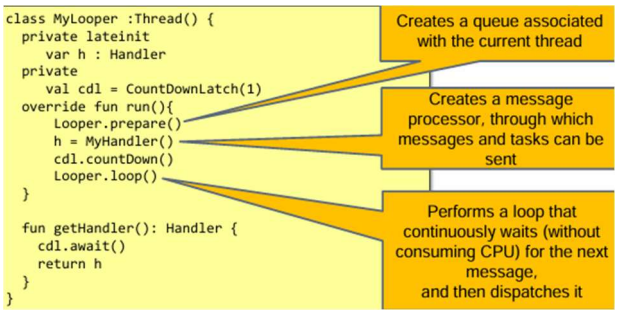


Reagisce ai messaggi in entrata.
Il **looper** è connesso ad una messageQueue e fetcha messaggi contenuti nella message queue.

Message queue può contenere 0+ messaggi, inserimenti in coda, esce da testa.

Il looper è connesso a 1+ **handler** che si occupano della gestione dei messaggi fetchati dal looper, processano messaggi.

Android.os.looper: classe contenente solo metodi statici



- **Looper.prepare**() → deve essere invocato da un thread per creare la coda associata
- Se un thread ha Looper.prepare, può creare 1+ **handler** in modo tale da gestire messaggi.
    - Se non crei handler, thread inutile perché non gestisce messaggi
- Dopo aver settato handlers, posso invocare **Looper.loop**()
    - Loop infinito che attende messaggi da altri thread
- Il thread che crea l'handler è quello che deve possedere la cosa sulla quale l'handler inserirà messaggi e dalla quale li prenderà. → è necessario che tutto sia creato dal thread secondario.
- **Cdl** → countdown lock → meccanismo semplice di sincronizzazione che fornisce due metodi
    - Awayt
    - Countdown

Quando il **thread è creato**, inizia immediatamente ad eseguire la run function (nel nuovo thread):

- Invoca Looper.prepare() → id del thread secondario, connesso a quello della message queue corrispodnente
- Handler creato
- Possibile countdown
- Eseguire Looper.loop

Un **thread esterno** che vuole interagire con il mio, deve ottenere l'Handler attraverso il metodo getHandler:

- Contiene cdl.awayt() che gestisce il lock, bloccando le richieste finchè cdl non diventa 0 (quindi finchè nessuno avrà il lock).

Androis.as.looper()

Generalmente il looper.loop method continua ad essere eseguito :

- Se non ci sono messaggi → non fa nulla
- Se c'è un messaggio → tira fuori e lo dispatcha all'handler da cui era stato inserito
    - Se ci sono più Handler, l'handler attraverso il quale il messaggio è stato inserito sarà quello attraverso il quale sarà gestito.
- Il messaggio sarà dispatchato all'Handler tranne se è il "quit message"
    - Quit message notifica al **looper** che è **terminato** attraverso:
        - quit() method
        - quitSafely() method

***Note ulteriori** (da slide):*

- *Generally, looper objects are only accessible from their own thread*
    - *Via method Looper.myLooper()*
- *Main looper is an exception*
    - *It is accessible from any thread, via Looper.getMainLooper()*

messageQueue:

- Unbounded linked list of messages to be processed by the consumer thread
  - Every looper (and thread) has at most one MessageQueue
- It is possible to retrieve the MessageQueue for the current thread via
  - Looper.myQueue()
- Solitamente FIFO
  - In alcuni casi posso insierire alcuni messaggi in testa se urgenti

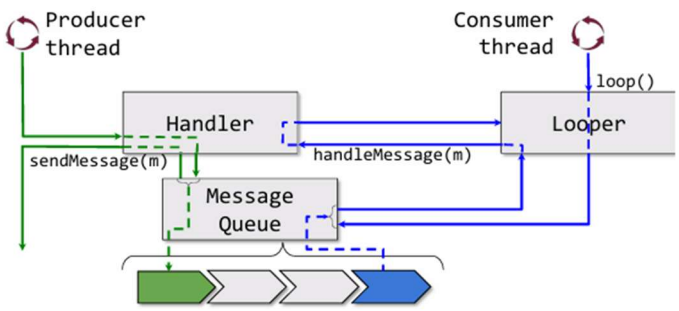
Andoird Handler:

2 tipi di messaggi supportati:

- **Runnables** → qualsiasi cosa che implementa l'interfaccia runnable → esegue metodo run()
- **Message objcect** → sarà processato dall'handler corrispondente nel suo handle message metod

Il messagio è composto da **3 interi e 1 oggetto**:

- **Command** → quale operazione deve il looper eseguire
- **One**
- **Two**
- **Any**
  Quando si definisce la semantica del messaggio si può definire cosa sono One e Two (Se sono qualcosa).



producer: inserisce messaggi nella coda,looper del consumer li tira via → fetcha e dispathca all'handler corrispondente.

Looper va avanti finchè non arriva il quit message.

Gestione problemi tipici:

Solo il thread **main** può accedere all'interfaccia **grafica**

- Se un thread secondario prova ad operare su di essa → errore exception

Sollo i thread secondari possono collegarsi alla rete

Serve un meccanismo che permette al thread secondario di notificare i suoi risultati al main thread.

- Android fornisce 5 strategie diverse.

Mandare messaggi al main thread:

- **Activity.runOnUiThread**:
  - Se si ha un reference all'attività stessa
  - Nota: stai prolungato la sua vita
- **View.postRunna**ble → esegue direttamente
- **View.postDelayed** → esegue con un ritardo
  - *Es: animation*

- `activity.runOnUiThread(r: Runnable)`
  - When this method is executed, if the current thread is not the main one, a new message that encapsulates the runnable is inserted into the queue
  - When the message will be processed, the main thread will executed it
  - It requires the secondary thread to hold a reference to the current activity (which could be a problem, should it finish...)
- `view.postRunnable(r: Runnable)`
  - Inserts the Runnable object in the message queue
  - It can be called from any thread as long as the view is displayed inside a window

`view.postDelayed(r: Runnable, l: Long)`

- Similarly to the previous case, it inserts the object into the queue, but it will not be processed before an interval equal to the given number of milliseconds
- In this case, too, the view must be connected to a window and be visible

- metodi corrispondenti:

  - Use instances of the **`Handler`** class, created in the main thread, and queue messages via one of the following methods
    - `sendMessage(…)`
    - `sendMessageDelayed(…)`
    - `sendMessageAtTime(…)`
    - `sendMessageAtFrontOfQueue(…)`
  - Or request the execution of a runnable, via
    - `post(…)`, `postDelayed(…)`, `postAtTime(…)`, `postAtFrontOfQueue(…)`

- **handler thread**: implemetazione del pattern che permette di definire un Handler creando un thread dandogli un nome e dopo averlo startato gli chiedi di creare un handler passandogli uhn handler al looper e specificando come si deve comportare quell'handler
  - Incorpora un looper e una messageQueue
- **Volendo si possono creare sottoclassi**

```
val ht= HandlerThread("HT");
ht.start();
val handler = Handler(ht.looper) {
    when (it.msg) {
       // Process messages here
    }
    return true
}
```

```
class MyHandlerThread(name:String): HandlerThread(name) {

    private val cdl = CountDownLatch(1)

    val handler : MyHandler by lazy {
        cdl.await()
        MyHandler(looper)
    }

    override fun onLooperPrepared() {
        cdl.countDown()
    }
```

Subclassing: con implementazione per prendere il proprio handler, con implementazione con countdown

Un thread esterno ottiene handler attraverso getHandler → in questo modo posso inserire e processare nuove richieste.