Background Operations

Short Recap

Fino ad ora, la complessità dell'applicazioni *Android* presentate accomunava un'insieme di *activities*, usufruendo degli *intents* affinchè le stesse possano dialogare, delle *views* per la visualizzazione grafica dell'applicativo ed infine degli *events* per manipolare le interazioni con l'utente.

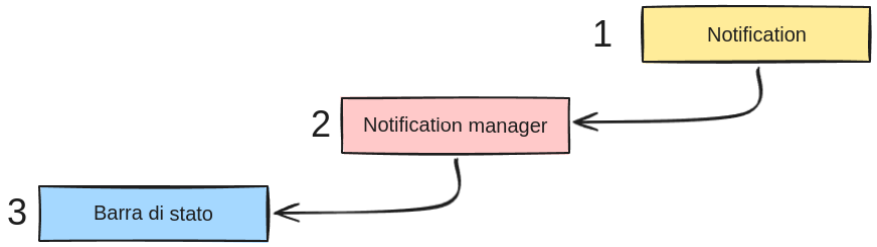
Tuttavia i concetti, brevemente descritti prima, sono eseguiti in *foreground*; occorre quindi accertarsi di cosa accada al di sotto.

Notifications

Le **notifiche** sono messaggi provenienti dall'applicazione *Android* sviluppata, tipicamente adottati per informare l'utente di certi avvenimenti. Si distinguono in due tipologie, a seconda dell'azione che si voglia conseguire:

- **Informativa passiva**, messaggio disposto a schermo per la sola visualizzazione
- **Informativa attiva**, da cui è possibile aprire l'applicazione oppure eseguire direttamente alcune operazioni

Tutte le volte in cui è creata una *notifica* è obbligatorio associarla ad un **canale comunicativo**, ciò avviene per due motivazioni principali. La prima ragione definisce la volontà dell'utente, stabilendo che tipo di messaggi istantanei voglia visualizzare, mentre la seconda è dettata dalla facilità di controllo mediante le impostazioni di sistema.



L'immagine riportata definisce appropiatamente il meccanismo di funzionamento delle *notifiche*. Di seguito si riportano gli step che caratterizzano il titolo esemplificativo.

- **Notification**, esecuzione della *notifica* tramite l'applicazione, attuando un **PendingIntent**, ossia un gancio per acquisire la *callback* di riferimento
- **Notification Manager**, componente del sistema *Android* responsabile della manipolazione della *notifica* e della disposizione a schermo della stessa all'interno della *status bar*
- **Status bar**, visualizzazione della *notifica*

Per implementare ed inviare una notifica occorre sottostare a specifici step, cosi definiti.

- **Primo step**, ottenere una referenza circoscritta al **NotificationManager**, dove in linguaggio *Kotlin* si traduce in:

```
val notificationManager = NotificationManagerCompat.from(this)
```

In breve, si tratta di un'invocazione del *System Service*, affinchè sia possibile comunicare al *sistema operativo* dell'esecuzione di un'operazione che potrebbe avere delle conseguenze sull'intero dispositivo

- **Secondo step**, creare un **canale di notifica**, attuato per le medesime motivazioni narrate precedentemente.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
  val CHANNEL_ID = "My Channel Id"
  val importance = NotificationManager.IMPORTANCE_DEFAULT
  val channel = NotificationChannel(CHANNEL_ID, "MyChannelName", importance)
  channel.description = "My description"
  val notificationManager = NotificationManagerCompat.from(this)
  notificationManager.createNotificationChannel(channel)
}
```

- **Terzo step**, costruire il corpo del messaggio, mediante l'utilizzo del *design pattern Builder*, in cui è suddivisa la costruzione dell'oggetto dalla propria visualizzazione grafica.

```
val builder = NotificationCompat.Builder(this, CHANNEL_ID)
  .setSmallIcon(androidx.core.R.drawable.notification_bg)
  .setContentTitle("Remember that you will die!")
  .setContentText("Let me explain a number of reasons why this is the case,
      blah, blah, blah...")
  .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

- **Quarto step**, attuare il *build* della *notifica* ed eseguirla.

```
val myNotificationId = 0
notificationManager.notify(myNotificationId, builder.build())
```

Nonostante, non è stato ancora approfondito il tema relativo a *notifiche* che siano caratterizzate da un *atteggiamento attivo*; ossia la possibilità di eseguire direttamente alcune operazioni dopo aver interagito con le comunicazioni visualizzate nella *status bar*.

A livello di codice è attuato un *Pending Intent*, il quale contiene un apposito *collante* che possa essere lanciato da componenti oppure elementi esterni dall'applicazione sviluppata.

```
val newIntent: Intent = Intent(this, MainActivity.javaClass)
newIntent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
newIntent.putExtra("caller", "notification")
val pendingIntent: PendingIntent = PendingIntent.getActivity (
  this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE
)
```

Concludendo è importante sottolineare che tutte le notifiche sono eseguite su molteplici *thread*, gestite dal *sistema operativo*; la volontà di adeguare un meccanismo simile è dettata dalla potenziale possibilità che le *notifiche* possano bloccare il device.

Multithreading

Un'ottima pratica di programmazione *Android*, prevede di assegnare una qualsiasi operazione, che richieda più di qualche millisecondo prima di terminare, ad uno specifico *thread*. Infatti, tendenzialmente, ogni attività è eseguita sul *main thread* assieme all'*User Interface*;

in assenza di una diversificazione, il contesto descritto potrebbe provocare un inadempimento dell'applicazione sviluppata.

Per rispondere a tale necessità, le operazioni accomunate da una caratterizzazione simile a quanto riportato, sono eseguite su *flussi di esecuzione* indipendenti; specificati all'interno del *Manifest* oppure definiti a livello di codice durante il *runtime*. Proseguendo, in questo modo, il *sistema operativo* possiede la totale libertà di gestire e manipolare i molteplici processi *runnati*.

In *Android* esistono tre metodi principali per la creazione di *thread*, suddivisi come segue:

- **Extend Thread**, estendere la classe *Thread*, in cui all'interno del metodo *run()* è riportato il comportamento a cui il *flusso di esecuzione* deve sottostare. Infine, istanziato un oggetto della classe, è richiamata la funzione *start()* per decretarne l'esecuzione.

```
public class MyThread extends Thread {
  public MyThread() {
    super("My Thread");
  }

  public void run() {
    // do your stuff
  }
}
```

```
MyThread m = new MyThread();
m.start();
```

- **Interface Runnable**, implementare l'interfaccia *Runnable*; tuttavia questo approccio richiede che sia presente un **Thread Pool**. Un *Thread Pool* è un insieme preallocato di *flussi di esecuzione* che eseguono *task* parallelamente, prelevate da una queue di riferimento.

  In questo modo, nuove mansioni sono eseguite da *thread* esistenti, evitando di inizializzare un numero spropositato di *flussi di esecuzione*.

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

```
executorService.execute(new Runnable() {
  @Override
  public void run() {
    // do your stuff
  }
});
```

Message Passing

Il *main thread* è incaricato di selezionare e porre all'interno della *UI* i molteplici componenti grafici; nessun *flusso di esecuzione* secondario dovrebbe avere una mansione simile, poichè potrebbero generare *race condition*. Tuttavia, in differenti casistiche, alcuni *worker threads* potrebbero avere la necessità di comunicare il risultato ottenuto una volta terminata la propria funzione.

È necessario stabilire delle comunicazioni fra loro, evitando di scatenare race condition

derivanti da un'erronea gestione della *concorrenza*.

Di seguito sono riportate tutte le componenti necessarie per riuscire nell'intento, suddivise come da titolo esemplificativo sottostante.

- Tutti i *thread* che vogliano ricevere dei risultati oppure delle comunicazioni, attuano un **handler**. Un **handler** è caratterizzato da una funzione *callback*, in modo tale che possa processare tutti i *messaggi in entrata* mediante un proprio *flusso di esecuzione*. Proseguendo, occorre che sia sviluppato un ulteriore oggetto, il quale consiste nella figura del **looper**, incaricato di conservare tutte le *comunicazioni* all'interno di una **queue**.

  A livello di codice, il comportamento descritto si traduce in:

```
public void run() {
  Looper.prepare(); // istanza la queue dei messaggi

  handler = new Handler(Looper.myLooper()) {
    @Override
    public void handleMessage(Message msg) {
      // do your stuff
    }
  }

  Looper.loop() // in attesa di ricevere messaggi
}
```

- Introdotta la sezione relativa al *receiver*, occorre definire l'atteggiamento del *sender*. Brevemente, il *thread* in questione dovrebbe ottenere una referenza del *ricevitore* ed inviargli il risultato.

  Per cui, considerando il *flusso di esecuzione* precedente oggetto della classe *MyThread*, il concetto si sviluppa come segue:

```
MyThread myThread = new MyThread();
myThread.start();
Handler myHandler = myThread().handler; // acquisizione del handler
```

  In questo modo, sarà possibile inviare un messaggio consecutivamente manipolato dal metodo *handleMessage()*.

```
Message m = myThread.handler.obtainMessage(); // nuovo messaggio inviato al
    mHandler del thread in attesa
m.arg1 = "Ciao, come stai?";
myThread.handler.sendMessage(m);
```

Coroutines

Gestire la *concorrenza* e le *comunicazioni* tra molteplici *flussi di esecuzione* non è banale e potrebbe scatenare differenti complessità implementative.

Android, assieme a *Kotlin*, mette a disposizione un ambiente di programmazione più semplice

rispetto agli esempi mostrati fino ad ora, dettato dalla presenza delle **Coroutines**.

Una *Coroutine* è un'istanza di una *computazione sospendibile*. Il suo atteggiamento è simile a quello di un *thread*, in grado di eseguire sezioni di codice. Inoltre non è associata ad alcun *flusso di esecuzione* specifico, può essere sospesa ed assegnata ad uno qualsiasi.

La definizione riportata esprime il concetto di **concorrenza strutturata**, ossia promuove la volontà di *Android* di ovviare a tutti i potenziali errori derivanti ad un erroneo codice implementato.

Nonostante la similitudine, è presente un importante numero di caratteristiche che differenzia una *Coroutine* da un *thread*; contraddistinte in tre fattori principali, cosi suddivisi:

- **Coroutine Scope**, rappresenta un ambiente che tiene traccia dell'istanze di computazione create ed offre differenti azioni affinchè sia possibile interagire con le stesse
- **Coroutine Context**, indica l'insieme dei metadati associati alla *Coroutine* circoscritta; a titolo esemplificativo si evidenzia la presenza del *dispatcher*, colui che è incaricato di eseguire l'istanza computazionale
- **Coroutine Job**, manipola una *Coroutine*, memorizzando una sua referenza all'interno di una variabile d'istanza

```
val scope: CoroutineScope = CoroutineScope(Dispatchers.Main)

fun myBlockingFunction() { /* do your blocking stuff */ }

val job: Job = scope.launch(Dispatchers.IO) {
  delay(100) // function that blocks the coroutine, not the thread
  myBlockingFunction()
}
```

Di seguito sono descritti due punti principali dell'esempio proposto, contraddistinti in:

- Tramite lo *scope* viene creata e lanciata una *Coroutine*, ma non eseguita, tipicamente riportata all'interno del *Main thread*, si osservi la sintassi indicata dal *dispatcher*
- Mediante il *job* è richiamata la *Coroutine* lanciata precedentemente, la quale consiste in un oggetto *runnable*, affinchè sia possibile definire dove l'istanza computazionale debba essere eseguita. A titolo esemplificativo, la *Coroutine* è stata indotta all'interno di un flusso di esecuzione specializzato in operazioni di *lettura* e di *scrittura*

Tuttavia, in certe circostanze, non è stabilito a priori quale sia la sezione di codice sviluppato che potrebbe bloccare l'intera applicazione; perciò entra in gioco la keyword **suspend**. Questo approccio impone al *caller* di richiamare la funzione mediante una *Coroutine*. In questo modo, qualsiasi sia l'operazione erronea, sarà possibile interrompere l'esecuzione della *sezione di codice*, senza alcuna ripercussione ai danni della *User Interface*.

```
suspend fun myBlockingFunction (): String {
  return withContext(Dispatchers.IO) { /* Blocking Code */ }
}

/* launch a coroutine in the main thread */
scope.launch {
  delay(100)
```

```
  val result = myBlockingFunction() // execute it in a IO thread and wait here
      until it finishes
  /* Do stuff with result in the main thread */
}
```

Inoltre, grazie al meccanismo delle *Coroutines*, è possibile variare il *contesto computazionale* di un blocco di codice, in favore di azioni mirate per l'aggiornamento della *User Interface*.

```
/* launch a coroutine in the main thread */
scope.launch {
  withContext(Dispatchers.IO) { /* Do your database operations */ }
  withContext(Dispatchers.Main) { /* Update UI */ }
}
```

Similmente a quanto avviene tramite le *promise* di *JavaScript*, è possibile effettuare della chiamate *asincrone* senza che le *Coroutines* siano sospese. Tuttavia, le keyword **async** e **await** possono essere attuate solamente all'interno dello *Scope*.

```
scope.launch {
  val deferred: Deferred<Unit> = async { myBlockingFunction() }
  /* CODE BLOCK A */
  deferred.await()
  /* CODE BLOCK B */
}
```

Rispetto allo sketch di codice proposto, la *Coroutine* non sarà, in questa particolare casistica, sospesa a causa della *myBlockingFunction()*, poichè richiamata *asincronicamente*; inoltre, è eseguita all'interno di un *thread separato*, pertanto il blocco di codice sottostante, *Block Code A*, sarà consecutivamente svolto all'interno del *Main thread*. Concludendo, data la presenza della keyword *await*, la computazione del *Block Code B* sarà eseguita solamente qualora sia restituito il risultato della funzione bloccante.

Services

Un **Service** è un componente in grado di eseguire lunghe operazioni in *background*. Rappresenta in *Android* un elemento simile alle *Activities*, ma carente di interfaccia grafica, infatti il suo scopo principe consiste nell'esecuzione di onerose operazioni.

Anch'esso deve essere dichiarato all'interno del *Manifest*, poichè è risvegliato mediante l'utilizzo degli *intents*.

```
<service android:name=".ExampleService" />
```

Un *servizio* garantisce un solido ambiente in grado di manipolare ed immagazzinare molteplici *flussi di esecuzione*; tuttavia, qualora le operazioni associate risultino estramamente onerose, dato che è contenuto all'interno del *main thread*, potrebbero provocare situazioni bloccanti per l'applicazione sviluppata. Di conseguenza, per attività che possano richiedere più di qualche millisecondo, occorre necessariamente utilizzare *threads* specifici.

Contrariamente alla sezione *XML* precedente, è possibile creare un *Service* da codice, attuando la seguente sintassi:

```
startService(intent) // tipicamente un intent esplicito
```

Proseguendo, per terminare la sua esecuzione, nuovamente a livello di codice, si utilizzano le istruzioni:

```
stopSelf() // all'interno dello stesso servizio
stopService(intent) // da ulteriori componenti
```

Broadcast Receiver

Un **Broadcast Receiver** è un componente che si attiva solamente quando accadono determinati *eventi*; in questa sezione il termine **event** è adeguato come sinonimo di *intent*.

Tipicamente è dichiarato all'interno del *Manifest*

```
<application>
  <receiver class="SMSReceiver">
    <intent-filter>
      <action android:value="android.provider.Telephony.SMS_RECEIVED" />
    <intent-filter>
  </receiver>
</application>
```

oppure definito a livello di codice

```
receiver = new BroadcastReceiver() {...}

protected void onResume() {
  registerReceiver(receiver, new IntentFilter(Intent.ACTION_TIME_TICK));
}

protected void onPause() {
  unregisterReceiver(receiver);
}
```

La differenza principale è dovuta al fatto che, se definito a livello di codice, non potrà essere eseguito qualora l'applicazione sia chiusa.