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.

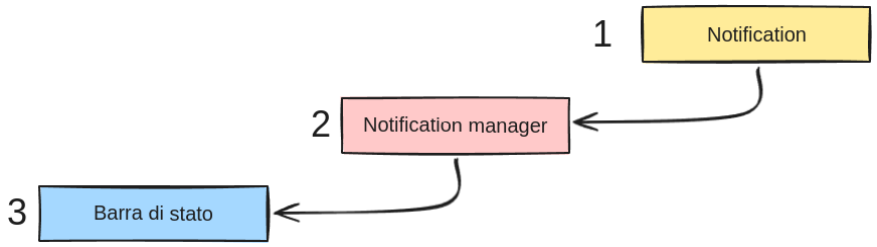
Tuttavia i concetti, brevemente descritti prima, sono eseguiti in *foreground*; occorre quindi accertarsi di cosa accada al di sotto.

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

...