Activity

Le **activities** rappresentano una parte fondamentale all'interno di *Android Studio*, data la loro importanza in questo contesto. Spesso le *activities* sono raffigurate come il primo contatto tra il sistema software realizzato e l'utente, da cui sono sviluppati metodi o funzioni in grado di poter acquisire dati provenienti da agenti esterni. Articolando l'introduzione, le stesse sono note anche secondo la denominazione **screen state**, ossia un fotogramma in grado di intrappolare tutto ciò che riesca a contraddistinguere il preciso momento in cui si trovi lo screen del dispositivo.

Concludendo *Android* stabilisce la sua operatività mediante uno **stack** di activity, infatti le medesime applicazioni si basano sul susseguirsi di insiemi di *attività*.

Le **activity** devono essere dichiarate all'interno del **Manifest**, contenitore di informazioni estremamente importanti a livello di sistema operativo; pertanto, occorre che siano descritte all'interno del Manifest prima che siano runnate, anche se solitamente, a seconda della versione di *Android Studio*, tutto ciò è definito automaticamente.

```
<application ...>
  <activity android:name=".MainActivity" android:exported="true">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

Dichiarazione di un'activity all'interno del Manifest

Il *Manifest* indica ciò che il sistema operativo possa visionare in riferimento all'applicazione che si trovi in stato *running*; proprio per questa principale motivazione, la sezione *XML* riportata precedentemente, definisce quali e come utilizzare le attività formulate al suo interno.

Activity Lifecycle

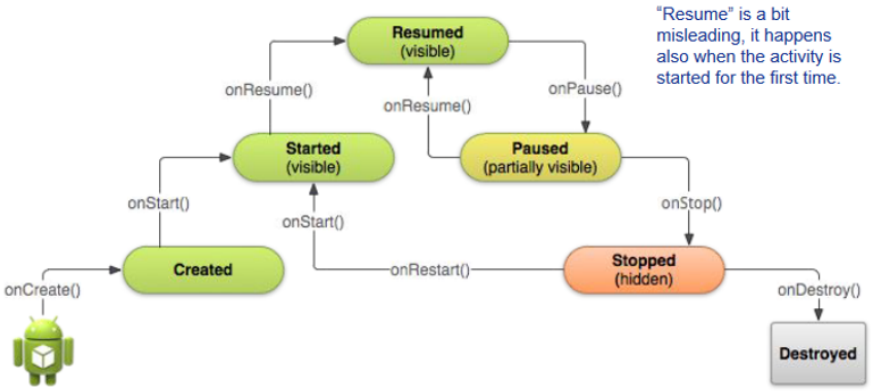
Le **activity** nel corso del loro *lifecycle*, attraversano una sequenzialità di *stati*, adottati per differenti contesti, soprattutto data la volontà di distinguere quale delle differenti operazioni debba essere svolta. Infatti l'operazione, o meglio definito *elemento computazionale*, raffigura l'unico item in grado di mantenere in vita l'applicazione circoscritta, per cui una gestione dei propri stati indica una sezione imprescindibile.

Come da immagine sottostante, si evidenziano un totale di cinque stati, di cui *destroyed* non è da considerarsi tale. In relazione, si prosegue attraverso una semplice descrizione di ognuno di essi, la lettura scaturisce sin dal momento in cui l'attività non risulti essere in memoria.

- **Created**, in questo istante l'*activity* non risulta ancora visibile, si tratta di una fase precedente al *running*, ma non alla compilazione
- **Started**, stato successivo in cui l'attività è visibile ma non ancora funzionante, o meglio non è interagibile, non riesce ancora a comunicare ed acquisire eventi generati da fattori

esterni

- **Resumed**, l'*activity* è in "primo piano", del tutto funzionante e pertanto interagibile
- **Paused**, si verifica l'*overlaid* di altre attività, un esempio concreto potrebbe essere un'*interrupt* generata da uno specifico evento caratterizzato da una maggiore priorità, il quale necessita di assimilare un livello e numero superiore di risorse, da cui scaturisce il fenomeno che provocherà lo stato descritto
- **Stopped**, in comune allo stato precedente, l'attività non può eseguire alcuna sezione di codice oppure ricevere dati, in aggiunta non risulta più visibile poichè portata in "secondo piano"



Detto ciò, l'insieme di caratteristiche riportate è dovuto all'avvenire di certi eventi, per lo più provocati da specifici metodi; secondo questa breve deduzione, si intuisce la presenza di molteplici funzioni che possano provocare i differenti passaggi di stato. Anche in questa circostanza suddivisi in:

- **onCreate()**, funzione richiamata nel momento in cui l'*activity* viene creata; solitamente contiene le sezioni di codice necessarie per imbastire l'intera logica dell'attività, pertanto sono eseguite solamente ad una occorenza. Inoltre è definito come il principale responsabile per la designazione della *user interface*
- **onStart()**, metodo successivo a *onCreate()*, interpellato poco prima che la finestra principale sia visibile all'utente e pertanto interagibile
- **onResume()**, come avviene per lo stato descritto prima, la funzione circoscritta definisce tutti i componenti affinchè l'*activity* possa essere *runnata*; terminata la propria computazione l'attività potrà essere eseguita
- **onPause()**, interviene qualora l'attività dovesse essere interrotta, a livello esemplificativo si evince attraverso un'interrupt a livello di priorità di processi; tipicamente non consegue con alcuna memorizzazione, data la sua estrema velocità di passaggio, tuttavia permette allo scheduler della CPU di rimuovere tutti i processi relativi all'attività affinchè non consumino risorse computazionali. Rispetto a quanto detto, l'attività risulta ancora visibile ma solo parzialmente, ossia tendenzialmente è applicata una velatura opaca per simboleggiarne l'inattività
- **onRestart()**, adottato solo quando l'*activity* sia momentaneamente sospesa oppure stoppata

- **onStop()**, l'attività non è più visibile e di conseguenza interagibile, spesso fenomeno causato da due fattori specifici; la prima causa potrebbe essere dedotta da uno "spostamento" in *background* di un'activity in favore di una corrisposta, oppure l'attività è in procinto di essere distrutta
- **onDestroy()**, funzione attuata qualora il sistema necessiti dello spazio in memoria oppure, con maggiore semplicità, qualche elemento ha chiamato il metodo circoscritto sulla precisa attività

Tuttavia è bene definire che i molteplici passaggi di stato sono controllati in minima parte dallo sviluppatore, da cui si evince l'importanza di saper almeno riconoscere il comportamento di un'attività durante il proprio *lifecycle*.

Logs with LogCat

Questa breve sezione si concentra sulla tematica denominata **logcat window**, usufruito come un vero e proprio strumento che possa aiutare qualsiasi sviluppatore durante l'implementazione di certe funzionalità, attuato principalmente durante attività di *debugging*; ad esempio potrebbero essere aggiunti alcuni messaggi all'applicazione sviluppata mediante l'estensione della *classe log*.

La *classe log* predispone un'elevato numero di metodi che possano definire l'importanza del messaggio di log visualizzato.

```
Log.v ("LABEL", "message")     // VERBOSE
Log.d ("LABEL", "message")     // DEBUG
Log.i ("LABEL", "message")     // INFORMATION
Log.w ("LABEL", "message")     // WARNING
Log.e ("LABEL", "message")     // ERROR
Log.wtf ("LABEL", "message")   // SHOULD NEVER HAPPEN IN LIFE
```

Dichiarazione dei log

Recreating Activities

Qualora si navigasse tra le differenti *activities* che compongono l'applicazione in questione, il sistema, tendenzialmente, adotta un approccio distruttivo dell'attività per poi, successivamente, crearne una nuova istanza. Un andamento simile provoca un sovraccarico a livello operativo, necessita uno strumento in grado di ricreare l'attività senza che essa sia sottoposta all'iter precedente. In tal senso è adoperato un *bundle*, denominato **instance state**, responsabilizzato della memorizzazione delle *activities*, da cui l'applicativo avrà la possibilità di riesumarne l'oggetto ogni qual volta ne avesse la possibilità.

Il salvataggio dei dati circoscritti avviene tramite l'*override* di due metodi principali, definiti come segue:

onSaveInstanceState(), generalmente metodo richiamato poco prima che l'*activity* sia posta in *stopped*

```
static final String STATE_SCORE = "playerScore";
@Override
public void onSaveInstanceState (Bundle savedInstanceState) {
```

```
  super.onSaveInstanceState (savedInstanceState);
  savedInstanceState.putInt(
    STATE_SCORE, mCurrentScore
  );
}
```

Salvataggio dei dati poco prima dello stato stopped

onRestoreInstanceState(), funzione richiamata poco dopo che l'attività sia posta in *started*

```
@Override
public void onRestoreInstanceState (Bundle savedInstanceState) {
  // Call the superclass to restore the views
  super.onRestoreInstanceState (savedInstanceState);
  mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
}
```

Salvataggio dei dati poco dopo dello stato started

In alternativa, i due metodi elencati, possono essere adeguati anche durante la fase di creazione dell'*activity*.

Task e BackStack

Le attività, durante il lifecycle di un'applicazione, concorrono tra di loro, posizionandosi visivamente al di sopra. In questo caso, l'attività precedente, ossia da poco sostituita da una propria corrisposta, viene salvata all'interno del **BackStack**

Il *BackStack* opera come una pila, pertanto secondo il funzionamento *FIFO*; navigando tra le differenti componenti dell'applicazione l'utente estrae l'attività corrente all'interno dello stack, distruggendola, e ricrea l'*activity* posizionata al di sopra della pila.

Tuttavia, potrebbe sorgere un problema legato all'esecuzione della stessa attività in due differenti fasi della medesima storyline, provocando, in tal contesto, la creazione di due differenti istanze; un esempio è dettato dall'utilizzo della mail-box, in cui si evidenzia un'*activity* usufruita per la visualizzazione del contenuto e una seconda adottata per la scrittura di un messaggio, per cui stessa attività ma due istanze differenti. Per evitare un caso simile, è possibile forzare il sistema affinchè non crei due istanze della stessa classe, mediante l'impiego di *flag*.

Come già accenato, navigare tra le differenti componenti obbliga lo sviluppatore a gestire le molteplici *activities*, spostando la **task** corrente in secondo piano.
Una *task* rappresenta un'unità coesiva, contenitrice di un *BackStack*, posizionata in primo piano oppure in secondo piano, a seconda che l'attività sia in stato *running* oppure *stopped*.

Generalmente la modellazione di task avviene per precise funzioni, da cui per ogni attività è possibile svilupparne il comportamento, modificando un'elevata molteplicità di parametri; contrariamente, eseguire un'applicazione dalla schermata home del dispositivo definisce sempre la stesso approccio.

Contexts

Un'attività è fortemente connessa ad un contesto, descritto come un punto di interazione tra app e sistema operativo, da cui qualunque entità ha la possibilità di richiamare le sue funzioni.