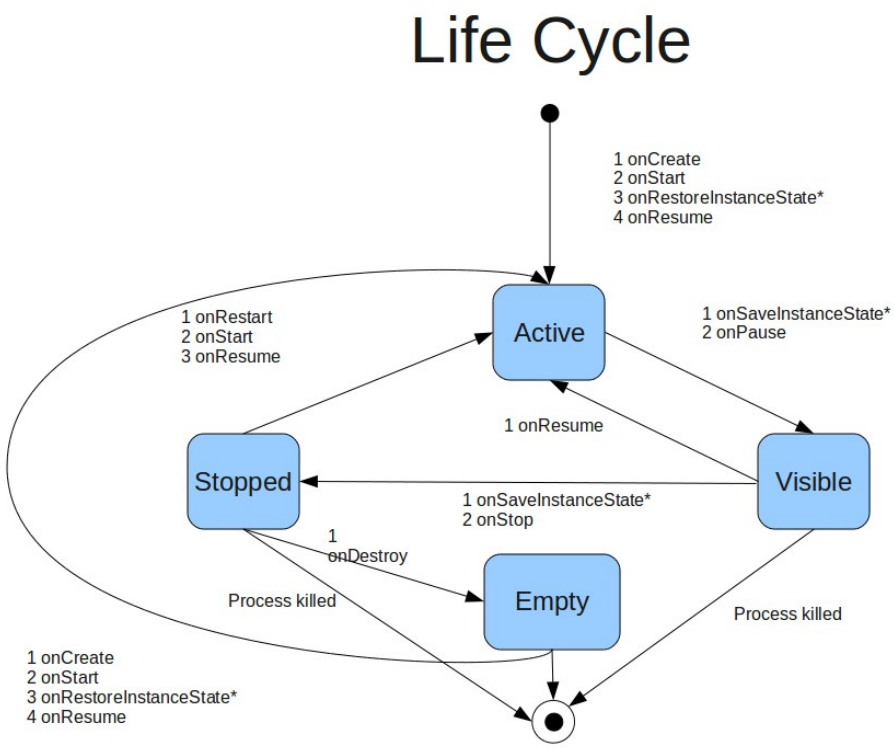
Lezione 21 – Applicazioni Android

Ciclo di vita delle Attività



Quando si sovrascrive uno dei metodi elencati in figura, come prima cosa bisognerebbe sempre chiamare il metodo sovrascritto.

Approfondimento sul ciclo di vita

Durante l'intero tempo di vita dell'attività che va dalla creazione alla distruzione, l'attività stessa entra una o più volte negli stati visibile e attivo. Ogni transizione da uno stato a un altro fa scattare l'invocazione dei gestori (handler) descritti nella figura. In particolare:

Il tempo di vita

Il tempo di vita dell'attività inizia alla prima chiamata del metodo `onCreate` e finisce all'ultima chimata del metodo `onDestroy`. E' possibile in alcuni casi che il processo dell'attività venga terminato senza una chiamata al metodo `onDestroy`.
Il metodo onCreate va utilizzato per inizializzare l'attivita: gonfiare l'interfaccia utente (ovvero estrarla dalla classe R per renderla visibile), allocare oggetti (in modo da poterli riusare in tutto il ciclo di vita), legare i dati ai relativi controlli e creare servizi e thread. Al metodo `onCreate` è passato un oggetto Bundle che contiene lo stato dell'interfaccia utente salvato dal metodo `onSaveInstanceState`. Nel metodo `onCreate` o nel metodo **`onRestoreInstanceState`** bisognerebbe sempre usare questo oggetto per ricreare la precedente interfaccia utente.

La sovrascrittura del metodo `onDestroy` dovrebbe rilasciare le risorse create da onCreate e assicurarsi che tutte le connessioni (connessioni di rete, connessioni a DB etc.) vengano chiuse.

Le linea guida per lo sviluppo di applicazioni su android raccomandano evitare la continua creazione e distruzione di oggetti (garbage collector). Per questa ragione si suggerisce di creare gli oggetti di utilità all'attività nel metodo `onCreate` e di riusarli nelle chiamate degli altri metodi.

Il tempo di visibilità

Il tempo di visibilità di un'attività inizia con la chiamata al metodo `onStart` e finisce con la chiamata al metodo `onStop`. Tra queste due chiamate l'attività è visibile all'utente anche se può non avere il fuoco ed essere parzialmente oscurata. Le attività possono diventare più volte visibili, anche se non probabile, Android può terminare attività nello stato visibile senza invocare il metodo `onStop`.
Il metodo `onStop` dovrebbe essere usato per mettere in pausa o fermare le animazioni, i theread, i listner dei sensori, il lookup del GPS, i timer, i servizi e ogni altro processo che viene usato esclusivamente per aggiornare l'interfaccia. (non serve a niente aggiornare un'interfaccia utente che non è vista dall'utente).
Si usa invece il metodo `onStart` o `onRestart` per riesumare i task fermati da `onStop`.
Il metodo `onRestart` è chiamata immediatamente prima del metodo `onStart` (a parte la prima volta in cui non viene chiamato). Quindi può essere usato per implementare eventuali comportamenti particolari quando l'attività ritorna visibile dopo essere stata oscurata.
I metodi `onStart/onStop` sono anche usati per registrare/cancellare i Broadcast Receiver che sono usati esclusivamente per aggiornare l'interfacia utente.

Il tempo di attività

Il tempo di attività inizia con la chiamata al metodo `onResume` e termina con la chiamata al metodo `onPause`.
Una Attività attiva è in primo piano e sta ricevendo l'input dall'utente. Un'attività probabilmente entrerà nello stato attivo più volte nel corso del suo tempo di vita in quanto verrà disattivata ogni volta che: 1) una nuova attività diventa attiva; 2) il dispositivo va in sleep; 3) l'attività perde il focus. Quindi bisogna cercare di rendere il codice dei metodi `onResume` e `onPause` relativamente veloci e leggeri.
Proprio prima di `onPause` viene chiamato il metodo **`onSaveInstanceState`**. Questo metodo deve provvedere a salvare lo stato dell'interfaccia utente nell'oggetto Bundle (che sarà poi passato ai metodi `onCreate` e `onRestoreInstanceState`**)**. Quindi in `onSaveInstanceState` di dovrebbe salvare lo stato dell'interfaccia (come lo stato dei checkbox, il focus dell'utente, il testo inserito ma non salvato etc.) in modo che l'utente si ritrovi la stessa interfaccia quando l'attività ritornerà attiva. **Si può assumere che i metodi `onSaveInstanceState` e `onPause` vengano chiamati prima che il processo dell'attività venga distrutto**. Quindi è importante sovrascrivere almeno uno di questi due metodi per salvare le modifiche perché rappresentano un metodo sicuro che viene chiamato prima della distruzione del processo.
Il metodo `onResume` normalmente dovrebbe essere abbastanza leggero e si occupa della sola riregistrazione di quei Broadcast Receivers che sono stati deregistrati dal metodo `onPause` (mentre il recupero dell o stato dell'interfaccia è garantito dalle chiamate dei metodi `onCreate` e `onRestoreInstanceState`).

Le Classi Activity speciali

La SDK Android include anche alcune sottoclassi di Attività che sono di comune utilizzo, alcune delle più usate sono:

- **MapActivity**: include la gestione delle risorse necessarie per visulizzare un mappa;
- **ListActivity**: una classe contenitore per attività che gestisce una ListView collegata ad una sorgente dati e che espone handler per ricevere eventi di selezione degli elementi;
- **ExpandableListActivity**: simile alla ListActivity ma espandibile;
- **TabActivity**: permette di includere più Attività o View in un singolo schermo con la possibilità di usare il tab per cambiare Attività o View selezionata.

Classe Applicazione

Nelle applicazion android possiamo sfruttare anche la classe `Application` per gestire meglio il ciclo di vita dell'intera applicazione. Estendendo la classe `Application` possiamo:

1. Mantenere lo stato dell'applicazione
2. Trasfererie oggetti tra componenti dell'applicazione (per esempio tra due attività della stessa applicazione)

Quando la sottoclasse di `Application` viene registrata nel manifesto, ne viene creata un'istanza quando il processo dell'applicazione viene creato. Per questa ragione questo oggetto si presta a essere gestito come singleton.
Una volta creata la classe che gestisce il ciclo di vita dell'applicazione bisogna registrare la classe nel manifesto specificando nel tag application il nome della classe nell'attributo `android:name`. Es.

```
<application android:icon="@drawable/icon"
android:name="MyApplication">
[... Manifest nodes ...]
</application>
```

Eventi del ciclo di vita dell'applicazione

La classe Application fornisce anche degli handler per gestire gli eventi principali del ciclo di vita dell'applicazione. Sovrascrivendo questi metodi si può personalizzare il comportamento dell'applicazione in queste circostanze:

- `onCreate`: è chiamato quando l'applicazione è creata e può essere usato per inizializzare il signelton e per creare e inizializzare tutte le variabili di stato e el risorse condivise.
- `onTerminate`: può essere chiamato (ma non c'è garanzia che venga chiamato se per esempio il processo viene terminato per liberare risorse) quando l'applicazione viene terminata.
- `onLowMemory`: permette di liberare della memoria quando il sistema ne ha poca disponibile. Viene generalmente chiamato quando i processi in secondo piano sono stati già terminati e l'applicazione attualmente in primo piano ha ancora bisogno di memoria.
- `OnConfigurationChanged`: viene chiamato quando c'è un cambiamento della configurazione. Al contrario delle attività che vengono terminate e fatte ripartire, all'applicazione viene notificato il cambiamento con la chiamata a questo metodo.

```
public class MyApplication extends Application {

private static MyApplication singleton;

// Returns the application instance
public static MyApplication getInstance() {
return singleton;
}

@Override
public final void onCreate() {
super.onCreate();
singleton = this;
...
}

@Override
public final void onTerminate() {
super.onTerminate();
...
}

@Override
public final void onLowMemory() {
super.onLowMemory();
...
}

@Override
public final void onConfigurationChanged(Configuration newConfig) {
super.onConfigurationChanged(newConfig);
...
}
}
```

Layout

I Layout sono delle classi che estendono la classe `ViewGroup` e permettono di gestire la disposizione di altri componenti detti componenti figli (estensione della classe View). I Layout possono essere nidificati permenttendo in questo modo la creazione di interfacce molto complesse.
Si può specificare il tipo di layout che è la root del documento. Tipi validi di layout predefiniti sono:

- `FrameLayout`: il più semplice fra i Layout che visualizza tutti i componenti figli nell'angolo in alto a sinistra. Se si aggiungono più figli, ogni nuovo figlio va a finire sopra il precedente nascondendo il precedente.
- `LinearLayout`: Allinea tutti i figli in una linea orrizontale o in verticale. Un Layout verticale ha una colonna di componeti, mentre un layout orrizontale ha una riga di componenti.
- `RelativeLayout`: il più flessibile dei Layout predefiniti permette di definire la posizione di ogni figlio relativamente agli altri e al bordo dello schermo.
- `TableLayout`: questo Layout permette di disporre i componenti in una griglia di righe e colonne.
- `Gallery`: Una Gallery visualizza una singola riga di componenti in una lista scorrevole orizzontalmente.

E' anche possibile creare controlli composti che estendono questi Layout predefiniti.

Esempio file layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello"
/>
<ListView android:layout_height="wrap_content"
android:id="@+id/listView1"
android:layout_width="fill_parent"></ListView>
<EditText android:layout_height="wrap_content"
android:id="@+id/editText1"
android:layout_width="fill_parent"
android:text="@string/textEdit"></EditText>
</LinearLayout>
```

Notiamo che per ogni elemento vengono usate le costanti `wrap_content` e `fill_parent` piuttosto che l'altezza o larrghezza esatta in pixel. In questo modo sfruttiamo al massimo la tecnica che ci permette di definire layout indipendente dalla dimensione dello schermo.
La costante `wrap_content` imposta la dimensione del componente al minimo necessario per contenere il contentuto del componente stesso mentre la costante `fill_parent` espande il componente in modo da riempire il componente padre.
Definire i Layout nei file xml diasaccoppia lo strato di presentazione dal codice Java e permette anche di creare Layout diversi in funzione dei vari hardware che sono caricati dinamicamente senza bisogno di modifiche al codice.

E' anche possibile implementare i Layout nel codice Java. In questo caso è buona norma utilizzare `LayoutParameters` da impostare con `setLayoutParams` method o passandoli al metodo `addView`.

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);
TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);
myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");
int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight,
lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight,
lWidth));
setContentView(ll);
```

Alcuni dei componenti predefiniti che possono essere visualizzati in un Layout (ma non solo):

- `TextView`: Una etichetta di testo in sola lettura. Supporta più linee, la foramttazione e il word wrapping..
- `EditText`: Un rettangolo per editare testo anche su più linee.
- `ListView`: Una lista verticale di elementi View.
- `Spinner`: Un componente composito che visualizza un TextView e una ListView associata che una volta selezionato un elemento della lista viene copiato nel TextView.

- `Button`: un pulsante.
- `CheckBox`: un pulsante a due stati che rappresenta lo stato tramite un quadrato che può essere spuntato o meno.
- `RadioButton`: un pulsante a due stati che se raggruppato a un numero arbitrario di altri RadioButton sono gestiti in modo tale che al massimo uno solo è selezionato.
- `ViewFlipper`: A ViewFlipper permette di disporre una collezzione di View in orizzontale in cui solo una View è visibile in ogni momento (la transizione traView è animata).

E' possibile personalizzare i componenti come vedremo nella prossima lezione.