

LAYOUT

In un progetto Android come sono organizzati i file per lo sviluppo di un'app? Elencare le principali cartelle/file e descrivere brevemente il loro contenuto.

L'organizzazione dei file per lo sviluppo di un'app è caratterizzata da varie directory. La prima cartella è la root, denominata "app", all'interno sono presenti le seguenti sottodirectory (l'ordine delle directory è importante):

- **manifests**: contiene il file **AndroidManifest.xml** che racchiude le informazioni essenziali dell'app;
- **java**: contiene il codice sorgente dell'applicazione, organizzato in vari packages, utilizzato per costruire l'app;
- **res**: contiene le risorse dell'app, ci sono al suo interno altre directory, come *"layout"* che fornisce la struttura statica dell'app (in xml), i *"drawable"* all'interno del quale verranno messi immagini e file multimediali, *"mipmap"* che contiene le varie icone dell'app a seconda della risoluzione del dispositivo, *"values"* contiene al suo interno file xml che forniscono metadati.

A cosa serve il file Manifest.xml?

Tutte le app Android hanno un file **AndroidManifest.xml** nella root del sorgente, è un file di risorse che contiene tutti i dettagli necessari dell'applicazione, ci consente di definire:

- **Packages**;
- **API**;
- **Librerie**;
- **Elementi base** dell'app, come activity e servizi;
- Dettagli sulle **autorizzazioni**;
- **Set di classi necessarie** prima del lancio.

Ci sono ulteriori informazioni che vengono riportate come:

- **Nome dell'applicazione**: titolo dell'app (*android:label= "nomeApp"*);
- **Icona dell'applicazione**: riferimento all'icona visualizzata nella schermata principale di Android per l'app;
- **Numeri versione**: è un singolo numero utilizzato per mostrare una versione dell'app (*android:versionCode= "1"*).

Nel file manifest.xml è possibile specificare un "intent-filter", ad esempio:

A cosa serve? Si dia sia una risposta generale (a cosa serve un intent-filter), sia una risposta per il caso specifico dell'esempio.

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

L'elemento **<intent-filter>**, figlio dell'elemento **<activity>** in un file manifesto di un'app, viene utilizzato per specificare i tre tipi di informazioni che quell'activity utilizza: **action**, **data** e **category**; questi tre parametri vengono utilizzati per la risoluzione implicita di un intent.

Per il caso specificato, abbiamo un'activity **ShareActivity** che può gestire l'invio (*action: android.intent.action.SEND*) di testo piano (*data: text/plain*); notiamo che tale activity ha la category di **DEFAULT**: questo indica che, quando viene effettuata la risoluzione implicita di un intent che indica come action e data rispettivamente *SEND* e *text/plain*, allora l'activity **ShareActivity** è quella di default per effettuare tale computazione.

Android prevede un meccanismo per il supporto di lingue diverse che rende molto facile cambiare la lingua utilizzata per i messaggi sullo schermo. Come funziona tale meccanismo?

All'interno della directory **res**/ ci sono sottodirectory per vari tipi di risorse, ad esempio file predefiniti come **res/values/strings.xml**, ma è possibile creare altre directory che aderiscono ad un determinato formato, come impostazioni internazionali dell'interfaccia utente. Quest'ultime conterranno file **strings.xml** i quali definiscono testi in lingue diverse. L'utente una volta che sceglie la lingua, verrà caricato il file **strings.xml** relativa alla scelta.

Il layout di un app può essere definito sia staticamente, tramite un file XML, che programmaticamente tramite istruzioni nel programma. Si discuta dei vantaggi e svantaggi e si faccia un esempio di un caso in cui è possibile usare solo uno dei due e non l'altro, motivando la risposta.

Un layout definisce la struttura dell'interfaccia utente, ad esempio di un'attività. Tutti gli elementi nel layout sono creati usando una gerarchia di oggetti **View** e **ViewGroup**. È possibile dichiarare un layout in due modi:

- **Dichiarare elementi dell'interfaccia utente in XML**: Android offre un semplice vocabolario XML che corrisponde alle classi e alle sottoclassi di visualizzazione, come quelle per **widget** e **layout**.
VANTAGGI: facile da specificare e separa in modo netto la definizione dell'UI dal codice dell'app (facile modifiche future).
SVANTAGGI: elementi statici.
- **Create un'istanza di elementi di layout in fase di esecuzione**: l'app può creare oggetti **View** e **ViewGroup** (e manipolarne le proprietà) a livello di codice.
VANTAGGI: dinamico, facilmente adattabile.
SVANTAGGI: gestione del layout nel codice dell'applicazione.

Un esempio in cui è possibile utilizzare soltanto uno dei due approcci riguarda, generalmente, casi in cui l'applicazione non prevede un'interazione con l'utente al runtime: più nello specifico, un'applicazione che deve mostrare semplicemente del testo all'utente (senza che egli debba interagire con l'interfaccia) può essere creata utilizzando esclusivamente l'approccio statico, attraverso i file XML.

Quali sono i widget standard per la costruzione dell'interfaccia utente? E quali sono i principali layout per il posizionamento dei widget? Per ognuno dei layout si descriva brevemente il meccanismo di posizionamento dei widget all'interno del layout.

I widget standard per la costruzione dell'interfaccia utente sono: **TextView, Button, EditText, ImageView, CheckBox, RadioButton**. I principali layout per il posizionamento dei widget sono:

- **LinearLayout**: organizza gli elementi in un'unica riga orizzontale o verticale ed è possibile creare una barra di scorrimento se la lunghezza della finestra supera la lunghezza dello schermo.
 - **RelativeLayout**: la posizione degli elementi è relativa al layout padre e agli altri elementi del layout, esempio figlio A alla sinistra del figlio B o allineato alla parte superiore del genitore.
 - **AbsoluteLayout**: consente di specificare posizioni esatte, esplicitando coordinate x e y, dei suoi figli. Sono meno flessibili.
 - **FrameLayout**: progettato per bloccare un'area sullo schermo per visualizzare degli elementi.
- Si possono creare Layout tramite un Adapter:
- **ListView**: visualizza un elenco scorrevole verticale.
 - **GridView**: visualizza una griglia scorrevole bidimensionale.

Ogni widget del layout prevede degli attributi che determinano parte della visualizzazione dello stesso widget all'interno dello schermo. Non tutti gli attributi possono essere applicati a tutti i widget; tuttavia ci sono degli attributi che si applicano a tutti i widget. Fra questi quali ritieni i più significativi o comunque i più utilizzati? Motivare la risposta.

Ogni elemento (**View** o **ViewGroup**) supporta degli attributi che specificano l'aspetto grafico, dove visualizzare l'elemento e fornire ulteriori informazioni. Ci sono degli attributi che sono comuni a tutti gli elementi, ad esempio:

- **"android:id"**: permette di assegnare un identificativo univoco agli elementi della UI. In questo modo, in una fase programmatica, si potrà ottenere il riferimento a quell'elemento, per poter modificare in modo dinamico lo stato dell'oggetto.
- **"android:layout_height"/"android:layout_weight"**: specificano altezza/larghezza di base della View, attributi obbligatori per qualsiasi View all'interno del layout. Ci sono varie unità di misurazione disponibili (px, dp, etc...) per entrambi.
- **"android:background"**: consente di definire il colore di sfondo di un widget (utile per verificare se è stato inserito nel layout).
- **"android:margin"**: utilizzato per creare spazio attorno agli elementi, al di fuori di qualsiasi bordo definito.
- **"android:padding"**: utilizzato per generare spazio attorno al contenuto di un elemento;
- **"android:gravity"**: specifica come un oggetto deve posizionare il suo contenuto dentro un componente;
- **"android:layout_gravity"**: specifica come un componente dovrebbe essere posizionato nel suo ViewGroup.

Cosa è e a cosa serve l'Android Virtual Device Manager?

Un **Android Virtual Device** è una configurazione di un dispositivo Android che si desidera simulare. **Android Virtual Device Manager** è un'interfaccia che si può avviare in Android Studio, il quale permette di creare e gestire un **AVD**.

Gli ambienti di sviluppo app per Android offrono la possibilità di usare un emulatore per eseguire le app. Quali sono i vantaggi e gli svantaggi dell'utilizzo di un emulatore?

Un emulatore duplica ogni aspetto del comportamento del dispositivo originale, sia hardware che software.

I vantaggi sono:

- Facilità di creazione di determinate situazioni, quali batteria scarica o arrivo di un messaggio
- Simulare un qualsiasi stato di rete
- Avviare una chiamata controllata

Gli svantaggi sono:

- Lento (a volte molto), alcune operazioni sono difficoltose
- È comunque un "simulatore"
- Possono esserci dei bug
- Un emulatore di dispositivo mobile non prende in considerazione fattori come surriscaldamento / drenaggio della batteria o conflitti con altre app (predefinite)
- Gli emulatori possono supportare solo determinate versioni del sistema operativo

EX: I real device hanno come vantaggio di essere veloci e l'input è facile da gestire (es. rotazioni display), inoltre l'esecuzione è reale.

Android permette al programmatore di specificare le dimensioni usando varie unità di misura:

- **dp, (density-independent pixels)**
- **sp, (scale-independent pixels)**
- **pt, points (1/72 di pollice)**
- **px, pixel reali**
- **mm, millimetri**
- **in, pollici (inches)**

Perché? Si discuta delle differenze fra queste varie possibilità.

Le varie unità di misura permettono al programmatore di adattare le interfacce utente ai numerosi dispositivi con risoluzioni diverse, ogni unità di misura è utile in diverse situazioni. Ad esempio:

- **Pixel**, permettono di definire elementi con dimensioni precise, indifferentemente dalla densità dello schermo
- **Pollici**, definiscono elementi in base alla dimensione dello schermo
- **Dp**, pixel indipendenti dalla densità, ovvero unità astratta che si basa sulla densità fisica dello schermo
- **Sp**, pixel indipendenti dalla scala, ovvero le dimensioni saranno regolate sia per la densità dello schermo che per le preferenze dell'utente impostate per il dispositivo.

A cosa serve la misurazione degli elementi grafici in "dpi" (density independent pixels). Si discuta del perché è necessaria e di come una cattiva gestione possa influire negativamente sull'aspetto grafico di un'app.

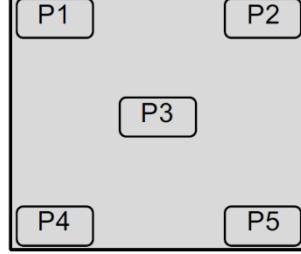
Per preservare la dimensione visibile dell'interfaccia utente su schermi con densità diverse, è necessario progettare l'interfaccia utente utilizzando **pixel indipendenti dalla densità (dp)** come unità di misura. Android traduce questo valore nel numero appropriato di pixel reali per ogni densità dello schermo.

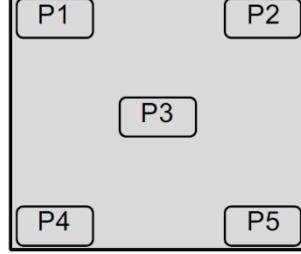
ES: Si considerino due schermi della stessa dimensione che hanno numero diversi di pixel. Se dovessi definire una vista larga "100px", apparirà molto più grande su uno dei due dispositivi, a seconda dei pixel a disposizione. Quindi si deve invece usare "100dp" per assicurarsi che appaia la stessa dimensione su entrambi gli schermi.

Si consideri un'app il cui layout prevede 4 pulsanti ai 4 angoli del display. Tale layout può essere implementato in vari modi. Descrivine uno (a parole, senza codice XML, spiegando solo come dovrebbe essere fatto il codice XML) mettendone in evidenza eventuali vantaggi o svantaggi.

Utilizziamo un **RelativeLayout** specificando per **width** e **height** il valore **match_parent** per prendere tutto il display a disposizione. Si comincia a creare il primo **Button** che deve essere posizionato in alto a sinistra. Ogni **width** e **height** viene egualato a **wrap_content**. Sfruttando il **RelativeLayout** impostiamo il "**alignParentLeft**" uguale a **true**, in questo modo il bottone verrà posizionato in alto a sinistra. Discorso simile per il bottone in alto a destra con la differenza che verrà usato "**alignParentRight**" egualato a **true**. Per il **Button** che verrà posizionato in basso a sinistra è necessario, oltre a mettere **alignParentLeft**, anche "**alignParentBottom**" sempre a **true**. Per quello posizionato in basso a destra sarà sempre necessario mettere **alignParentRight** e **alignParentBottom**, entrambi a **true**.

Scrivere un file di layout che implementi l'interfaccia riportata in figura. Il rettangolo è un frame di 200x200px e deve essere posizionato al centro del display. I riquadri con etichetti "Pn" sono dei pulsanti la cui pressione deve essere collegata alla funzione "pulsantePremuto"

<pre><RelativeLayout android:layout_width="match_parent" android:layout_height="match_parent" tools:context=".MainActivity"> <RelativeLayout android:layout_width="400dp" android:layout_height="400dp" android:layout_centerInParent="true" android:background="#A9A9A9"> <Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="P1" android:onClick="pulsantePremuto" /> <Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="P2" android:onClick="pulsantePremuto" android:layout_alignParentRight="true" /> <Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="P3" android:onClick="pulsantePremuto" android:layout_centerInParent="true" /> <Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="P4" android:onClick="pulsantePremuto" /> <Button android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="P5" android:onClick="pulsantePremuto" android:layout_alignParentBottom="true" android:layout_alignParentRight="true" /> </RelativeLayout> </RelativeLayout></pre>	
---	--

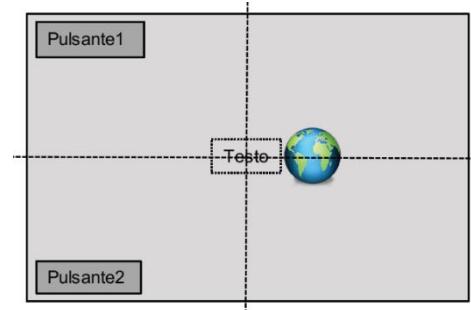


Nella seguente tabella sono riportati alcuni tipi di layout (sulle colonne) ed alcuni attributi (sulle righe). Indicare con una X all'incrocio fra riga e colonna gli attributi che hanno senso per lo specifico layout. Si fornisca anche una breve motivazione per le risposta inserite nella tabella.

	Linear Layout	Relative Layout	Grid Layout	Frame Layout
android:layout_gravity		•		
android:layout_centerVertical	•	•	•	•
android:id	•	•	•	•
android:orientation	•	•	•	
android:layout_toRightOf	•	•	•	•

Si scriva un frammento di codice XML che descriva il layout raffigurato a fianco scegliendo opportunamente gli elementi da utilizzare (le linee punteggiate non sono elementi del layout ma indicano il centro del layout).

```
<RelativeLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context=".MainActivity">  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Pulsante1"  
    />  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Pulsante2"  
        android:layout_alignParentBottom="true"  
    />  
  
    <TextView  
        android:id="@+id/TV"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_centerInParent="true"  
        android:text="Testo"  
    />  
  
    <ImageView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_toRightOf="@+id/TV"  
        android:src="@drawable/world"  
    />  
/</RelativeLayout>
```

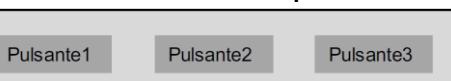


Si costruisca tale layout utilizzando un file statico che specifica i primi 2 pulsante ed inserendo dinamicamente il terzo pulsante (dettagliare il file XML e lo snippet di codice che serve ad aggiungere il terzo pulsante).

Si consideri il LinearLayout indicato in figura:

```
<LinearLayout  
    android:id="@+id/LL"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="horizontal"  
    tools:context=".MainActivity">  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Pulsante1"  
    />  
  
    <Button  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Pulsante2"  
    />  
/</LinearLayout>
```

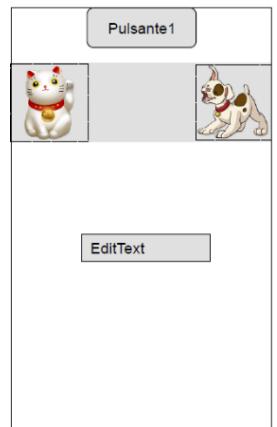
```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    Button b3 = new Button(this);  
    b3.setText("Pulsante3");  
  
    LinearLayout ll = findViewById(R.id.LL);  
    ll.addView(b3);  
}
```



Si completi il textoXML che descrive l'interfaccia utente specificata nel disegno a destra. Il RelativeLayout già presente si riferisce all'intero schermo.

```
<RelativeLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:background="#FFFFFF"  
    tools:context=".MainActivity">  
  
    <Button  
        android:id="@+id/B1"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_centerHorizontal="true"  
        android:text="Pulsante1"/>  
  
    <FrameLayout  
        android:id="@+id/FLimage"  
        android:layout_below="@+id/B1"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_marginTop="40dp">  
  
        <ImageView  
            android:layout_width="100dp"  
            android:layout_height="100dp"  
            android:layout_gravity="right"  
            android:src="@drawable/cat"/>  
    </FrameLayout>  
/</RelativeLayout>
```

```
<ImageView  
    android:layout_width="100dp"  
    android:layout_height="100dp"  
    android:layout_gravity="right"  
    android:src="@drawable/dog"/>  
/</FrameLayout>  
  
<EditText  
    android:layout_below="@+id/FLimage"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_centerHorizontal="true"  
    android:text="EditText"  
    android:layout_marginTop="100dp"/>  
/</RelativeLayout>
```



Si scriva un file di layout che determini l'aspetto grafico mostrato nella figura. Il riquadro esterno rappresenta tutto lo schermo con uno sfondo di colore grigio chiaro. La foto dell'allenatore è una singola immagine, mentre le foto dei componenti della squadra sono visualizzate in un gridView. Il colore di sfondo nella parte dei titoli è giallo, mentre il colore di sfondo del gridView è azzurro (per specificare i colori è sufficiente scrivere qualcosa del tipo @color/azzurro).

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="#808080">
    <TextView
        android:id="@+id/TVallenatore"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Allenatore"
        android:padding="10dp"
        android:textSize="20dp"
        android:background="#FFFF00"
        android:gravity="center"
    />
    <ImageView
        android:id="@+id/IVallenatore"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/baby_yoda"
        android:layout_below="@+id/TVallenatore"
        android:layout_margin="40dp"
        android:layout_centerHorizontal="true"
    />
```

```
<TextView
    android:id="@+id/TVsquadra"
    android:layout_below="@+id/IVallenatore"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Squadra"
    android:padding="10dp"
    android:textSize="20dp"
    android:background="#FFFF00"
    android:gravity="center"
/>
<GridView
    android:id="@+id/GVsquadra"
    android:layout_below="@+id/TVsquadra"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="10dp"
    android:background="#007fff"
    android:numColumns="3"
    android:gravity="center"
/>
</RelativeLayout>
```



Si scriva un file di layout che determini l'aspetto grafico mostrato nella figura. Il riquadro esterno rappresenta tutto lo schermo con uno sfondo di colore grigio chiaro, mentre il contenitore centrale con uno sfondo giallo ha un'altezza di 200dp. Il testo "email:" è visualizzato in un TextView. Di seguito c'è un EditText con il suggerimento "type here"

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#A9A9A9"
    tools:context=".MainActivity">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="200dp"
        android:layout_centerInParent="true"
        android:background="#FFFF00">
        <TextView
            android:id="@+id/TV"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:text="email: "/>
        <EditText
            android:layout_toRightOf="@+id/TV"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:background="#FFFFFF"
            android:hint="type here"/>
    </RelativeLayout>
</RelativeLayout>
```



LISTVIEW

Che cosa è un ListView? Che cosa è un Adapter? In che modo ListView e Adapter interagiscono?

Un **ListView** è un Widget specifico per le liste, raggruppa diversi elementi e li visualizza in un elenco scorrevole verticale. Gli elementi vengono automaticamente inseriti nell'elenco utilizzando un **Adapter** che estrae il contenuto da un'origine come un array o un DB.

Un Adapter è un oggetto, che può essere agganciato a qualsiasi **AdapterView** (ovvero ListView), che gestisce la visualizzazione degli elementi in un **AdapterView**: fornisce accesso a tali elementi ed è responsabile della creazione di una **View** per ogni elemento nel data set.

L'**Adapter** viene assegnata a **ListView** tramite il metodo **setAdapter** sull'oggetto **ListView**: `listView.setAdapter(arrayAdapter)`

Descrivere schematicamente tutto ciò che serve per far comparire sullo schermo un ListView (semplice) con una lista di nomi (definiti in un array). Si specifichi per ogni passo l'istruzione, il widget o il file XML, necessari per il ListView.

- Si definisce nel file **xml (activity_main)** un **widget ListView**, specificando l'id (mylistview) e i vari parametri:

```
<ListView  
    android:id="@+id/mylistview"  
    android:layout_width="wrap_content"  
    android:layout_width="wrap_content"  
/>
```

- Si crea un ulteriore file **xml** per rappresentare il **singolo elemento della lista**, coi relativi parametri:

```
<TextView  
    android:id="@+id/textViewList"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="" android:padding="10dp"  
    android:textSize="22dp"  
/>
```

- All'interno del file Java **MainActivity** si definisce un **Array di stringhe** contente nomi di persone:

```
String [] array = {"Pasquale", "Maria", "Michele", ...}
```

- Si definisce un **ArrayAdapter**, dove **context** rappresenta il contesto dell'applicazione, **R.layout.list_element** rappresenta il file di layout dove viene definito il singolo elemento nella lista, **R.id.textViewList** che rappresenta l'identificativo del TextView nel file di layout e poi c'è **array** che rappresenta le informazioni da visualizzare nel ListView:

```
ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(context, R.layout.list_element, R.id.textViewList, array);
```

- Viene individuato il **widget ListView**:

```
listView = (ListView) findViewById(R.id.mylistview);
```

- Infine si associa **l'adapter al widget**:

```
listView.setAdapter(arrayAdapter);
```

Che cosa è un Adapter customizzato per un ListView? Si fornisca una spiegazione possibilmente con frammenti di codice.

Un **Adapter customizzato** nasce dall'esigenza di creare una visualizzazione personalizzata di un **ListView**.

Un **Adapter** customizzato per un **ListView** è un **Adapter** che consente di adattare, ad un **ListView**, un elemento che contiene dei sottoelementi.

L'elemento da adattare dev'essere descritto in un file **XML**, in cui si inserisce il layout dell'elemento (compreso, pertanto, dei sottoelementi).

Per creare un adapter customizzato, occorre creare una classe che estende **ArrayAdapter**, al cui interno va aggiunto il costruttore (avente come parametri il **context**, la **resource** id dell'elemento da adattare, ed una **lista di elementi** da adattare) ed il metodo **getView()**: quest'ultimo viene invocatoognqualvolta è necessario mostrare un elemento del **ListView** (ad esempio, al lancio dell'applicazione o a seguito di uno scroll), ed è utilizzato anche per eseguire **l'inflate** del file di layout dell'elemento nel **ListView**.

Per adattare un **ListView**, nell'activity in cui tale lista è presente occorre seguire i prossimi step:

1. Creare l'adapter customizzato ed aggiungerlo al **ListView**:

```
CustomAdapter adapter = new CustomAdapter(getApplicationContext(), R.layout.?, new ArrayList<Object>);  
ListView listView = (ListView) findViewById(R.id.listView);  
listView.setAdapter(adapter);
```

2. Aggiungere i vari oggetti all'adapter:

```
for (...) { object = ...; adapter.add(object); }
```

3. Aggiungere, se richiesto, un listener sul **ListView**:

```
listView.setOnItemClickListener(new OnItemClickListener() {  
    @Override  
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {  
        ...  
    }  
});
```

Un listview prevede **OnItemClickListener** che gestisce i click sugli elementi della lista chiamando il metodo **onItemClick** al quale viene passato un riferimento dell'elemento selezionato. Se usiamo un listview customizzato, in cui ogni elemento della lista è composto da vari sottoelementi (es. una foto, un nome, un numero), il riferimento passato al metodo **onItemClick** non distingue quale dei sottoelementi è stato selezionato. Come si può fare per reagire in maniera diversa in funzione di quale dei sottoelementi è stato selezionato con il click?

Se usiamo un **ListView** customizzato, per distinguere quale sottoelemento è stato selezionato è opportuno definire dapprima un **Adapter customizzato**. Supponendo che esso sia stato già creato ed “agganciato” al **ListView**, allora tale problema deriva dal fatto che è stato passato un **OnItemClickListener** sull'intero elemento della lista, “creato” dal **CustomAdapter**. Per risolvere, è opportuno agganciare un onclick listener su ogni sottoelemento del **ListView**; inoltre, per far sì che si possa risalire alla posizione dell'elemento dell'adapter cliccato, è possibile utilizzare il metodo **setTag()** su ogni sottoelemento passando la posizione dell'elemento adattato (questo va fatto nel metodo **getView()** del custom adapter): questa potrà essere ricavata in un altro punto con il metodo **getTag()** sullo stesso sottoelemento.

Si consideri il seguente frammento di codice che definisce un **OnItemClickListener** per un **ListView**.

```
listView.setOnItemClickListener(new OnItemClickListener() {  
    @Override  
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {  
        String str = listView.getItemAtPosition(position).toString();  
        // Fai qualcosa con l'elemento ...  
    }  
});
```

Si spieghi il ruolo del parametro **position**.

Il metodo **onItemClick()** viene invocato quando si effettua un click su un elemento del listview. Si assume che sul **listView** è stato agganciato un semplice **ArrayAdapter<String>**. Il parametro **position**, al click, tiene traccia dell'indice dell'array che è stato adattato al list view dall'adapter: in questo caso, viene ottenuta, con la prima linea di codice del metodo **onItemClick()**, una stringa dell'elemento in posizione position del **listView**.

Il seguente frammento di codice serve a gestire un **listView** con gli elementi di un array. Completare il codice dell'**onItemClickListener** (riquadro vuoto) in modo tale che quando l'utente clicca un elemento visualizzato nel **listView** venga mostrato un **Toast** con il nome cliccato.

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    String [] nazioni = {"Italia", "Francia", "Spagna", "Germania", "Portogallo", "Svizzera", "Belgio", "Polonia", "Grecia", "Svezia"};  
    listView = (ListView)findViewById(R.id.mylistview);  
    ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(this, R.layout.list_element, R.id.textViewList, array);  
    listView.setAdapter(arrayAdapter);  
    listView.setOnItemClickListener(new OnItemClickListener() {  
        @Override  
        public void onItemClick(AdapterView<?> parent, View view, int n, long label) {  
            String str = listView.getItemAtPosition(position).toString();  
            // Show Toast  
            Toast.makeText(getApplicationContext(), "Nazione :" +str, Toast.LENGTH_LONG).show();  
        }  
    });  
}
```

Il seguente codice incompleto è un CustomAdapter per una lista customizzata di oggetti Object. Ogni oggetto Object possiede i getter getString() e getInt(). Il file di layout "list_element" contiene 2 TextView con i seguenti identificativi: "stringa" e "intero". Completare il CustomAdapter per creare la view di ogni singolo elemento.

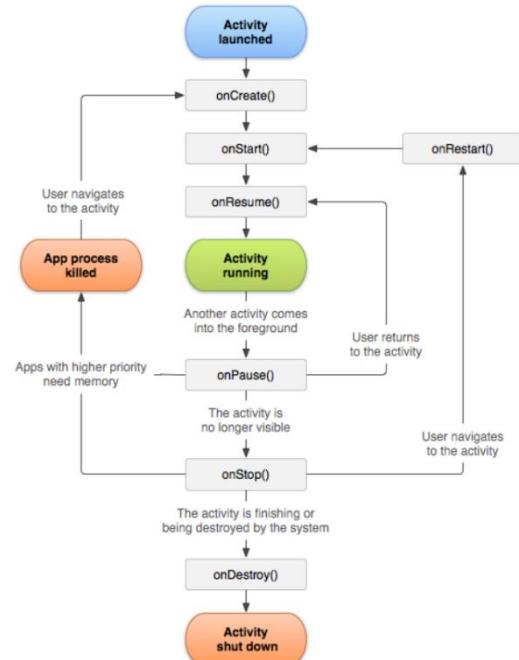
```
public class CustomAdapter extends ArrayAdapter<Object> {
    private int resource;
    private LayoutInflater inflater;
    public CustomAdapter(Context context, int resourceId, List<Objects> objects) {
        super(context, resourceId, objects);
        resource = resourceId;
        inflater = LayoutInflater.from(context);
    }
    @Override
    public View getView(int position, View v, ViewGroup parent) {
        if (v == null) {
            v = inflater.inflate(R.layout.list_element, null);
        }
        TextView tv1 = v.findViewById(R.id.stringa);
        TextView tv2 = v.findViewById(R.id.intero);
        Object o = getItem(position);
        tv1.setText(o.getString());
        tv2.setText(o.getInt());
        return v;
    }
}
```

CICLO DI VITA

Il ciclo di vita di un'activity prevede l'esecuzione dei seguenti metodi: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`. Indica per ognuno di questi metodi, o almeno per la maggior parte di essi, un'operazione che viene tipicamente implementata nel metodo.

In `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()` viene invocato il metodo della superclasse corrispondente senza parametri.

- **`onCreate()`:** invocazione del metodo `onCreate()` della superclasse con parametro l'oggetto Bundle e setta visibile il layout che viene descritto nell'activity_main.
- **`onStart()`:** iniziare a disegnare elementi visivi, eseguire animazioni, ecc
- **`onResume()`:** provare ad aprire dispositivi ad accesso esclusivo o per accedere a risorse singleton.
- **`onPause()`:** salvare qualsiasi stato persistente dell'attività in fase di modifica, per assicurarsi che non vada perso nulla se non ci sono risorse sufficienti per avviare la nuova attività senza prima averle ucciso e per fermare le cose che consumano una notevole quantità di CPU per rendere il passaggio all'attività successiva il più velocemente possibile.
- **`onStop()`:** per interrompere l'aggiornamento dell'interfaccia utente, l'esecuzione di animazioni e altre cose visive.
- **`onDestroy()`:** implementato per liberare risorse come thread associati a un'attività, in modo che un'attività distrutta non lasci tali cose mentre il resto dell'applicazione è ancora in esecuzione.



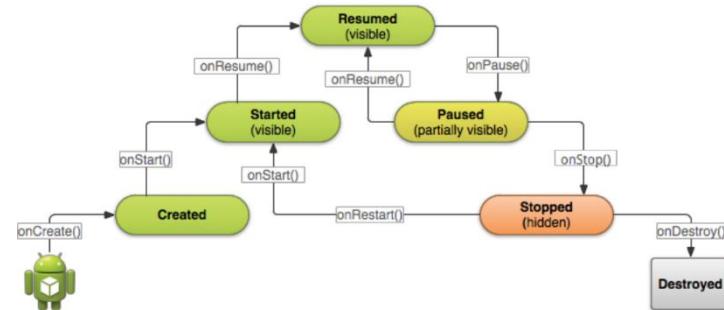
Si descriva il ciclo di vita di un'activity.

Ogni **activity** di un'applicazione possiede un ciclo di vita. L'activity può trovarsi in sei stati diversi: **Created, Started, Resumed, Paused, Stopped, Destroyed**. Quando l'app viene creata, il sistema operativo Android crea l'activity principale dell'app: viene invocato il listener `onCreate()` e l'activity passa allo stato **Created**, successivamente, l'activity viene lanciata e messa in **resume** (qui, l'utente potrà interagire con l'UI): viene invocato il listener `onStart()` e l'activity passa allo stato **Started**, poi viene invocato il listener `onResume()` e l'activity passa allo stato **Resumed**. Quando l'app lancia una nuova activity, quella corrente viene messa in pausa: si invoca il metodo `onPause()` e l'activity passa allo stato **Paused**; qui, si distinguono tre casi:

- 1) Il SO necessita di risorse, per cui distrugge momentaneamente l'activity e ritorna all'`onCreate()`;
- 2) L'utente ritorna nell'activity, per cui si invoca `onResume()` e l'activity passa allo stato **Resumed**;
- 3) L'utente chiude l'app, per cui l'activity viene definitivamente distrutta.

Dallo stato **Paused**, si può passare allo stato **Stopped** (prima s'invoca `onStop()`): qui possono accadere i tre casi precedenti, ma al caso 2 anziché passare allo stato **Resumed** l'activity passa allo stato **Started**, invocando i metodi `onRestart()` e `onStart()`. Infine, l'activity viene distrutta: si invoca il metodo `onDestroy()` e l'activity passa allo stato **Destroyed**.

Dal momento in cui un'Activity viene lanciata si può trovare in 5 diversi stati. Quali sono questi stati? Quali sono i metodi che permettono di passare dall'uno all'altro? Quali di questi stati sono "transienti" (cioè l'activity rimane per pochissimo tempo in questi stati) e quali invece più "duraturi" (cioè l'activity può rimanere anche per molto tempo in questi stati)?



Transienti: Created, Started, Paused

Duraturi: Resumed, Stopped

L'intero ciclo di vita va dall'esecuzione del metodo `onCreate()` all'esecuzione del metodo `onDestroy()`. Da quale a quale metodo vanno invece il periodo di "visibilità" ed il periodo di "foreground" dell'activity?

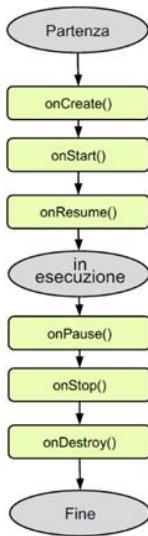
Visibilità: `onStart()` → `onStop()`

Foreground: `onResume()` → `onPause()`

Si disegni il ciclo di vita delle attività spiegando quali transizioni avvengono in modo automatico e quali invece per effetto di un intervento o dell'utente o del sistema.

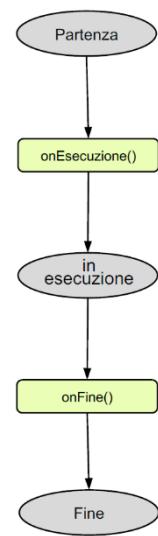
Automatico: `onStart()`, `onResume()`, `onStop()`, `onDestroy()`, `onRestart()`

Intervento utente/sistema: `onCreate()`, `onPause()`



Il ciclo di vita delle activity, riportato schematicamente a sinistra, prevede l'esecuzione in successione di 3 metodi (`onCreate`, `onStart`, `onResume`) per far partire l'esecuzione di un'app.

Perché? Non sarebbe stato meglio avere un solo metodo, come indicato nella figura a destra, nel quale eseguire tutto ciò che viene fatto nei 3 metodi `onCreate`, `onStart`, `onResume`?



Analogamente per distruggere un app è prevista l'esecuzione di 3 metodi in successione (`onPause`, `onStop`, `onDestroy`). Non sarebbe stato più semplice avere un solo metodo come indicato nella figura a destra? Motivare la risposta.

Ogni callback consente di eseguire lavori specifici appropriati per un determinato cambio di stato. Fare il lavoro giusto al momento giusto e gestire correttamente le transizioni rende l'app più solida e performante. Ad esempio, una buona implementazione dei callback del ciclo di vita può aiutare a garantire che l'app eviti:

- Arresto anomalo se l'utente riceve una telefonata o passa a un'altra app durante l'utilizzo dell'app.
- Consumo di preziose risorse di sistema quando l'utente non lo utilizza attivamente.
- Perdere i progressi dell'utente se escono dall'app e ci ritornano in un secondo momento.
- Arresto anomalo o perdita dei progressi dell'utente quando lo schermo ruota tra orientamento orizzontale e verticale.

Che cosa è il backstack? Supponendo che l'activity A sia l'unica in esecuzione (l'unica presente nel backstack), che la stessa app lancia una nuova activity B che a sua volta lancia l'activity C, che lancia l'activity D, quale è il backstack a questo punto? Cosa succede se dall'activity D si preme il pulsante di "back"? Cosa si deve fare se si vuole fare in modo che dall'activity D, si torni direttamente ad A quando si preme il pulsante di back?

Un task è una raccolta di attività con cui gli utenti interagiscono durante l'esecuzione di un determinato lavoro, disposte in uno stack denominato *backstack*. Lo stack ha una struttura LIFO e memorizza le attività nell'ordine della loro apertura. Le attività nello stack non vengono mai riorganizzate. La navigazione del backstack viene eseguita con l'aiuto del pulsante *back*.

La situazione nel backstack è la seguente: D, C, B, A. Se si preme back, ovvero si esegue un pop dallo stack dell'activity, l'attività D viene distrutta e C viene messa in foreground.

Se si vuole far in modo che dall'activity D si torni direttamente ad A quando si preme il pulsante di back, occorre lanciare le activity B e C in modo tale che non vengano memorizzate nel backstack: questo dev'essere effettuato programmaticamente, aggiungendo un ***FLAG_ACTIVITY_NO_HISTORY*** all'intent attraverso il metodo `intent.setFlags(int flag)`.

Si spieghi il meccanismo del backstack. In relazione a tale meccanismo che differenza c'è fra una activity e un frammento?

Il backstack è un meccanismo che conserva l'ordine di utilizzo delle activity di un'applicazione in una pila:

- all'apertura dell'applicazione viene effettuato un push() della main activity nel backstack, ed un altro push() viene effettuato all'avvio di nuove activity dell'applicazione;
- quando l'utente clicca sul tasto Back (oppure viene cliccato un qualsiasi pulsante che ottiene ciò), viene effettuato un pop() dell'activity corrente.

Le differenze sono:

Activity	vs	Fragment
Componente che fornisce un'interfaccia utente in cui l'utente può interagire		Fa parte di un'attività, che contribuisce all'interfaccia utente di tale attività
In <i>landscape</i> possiamo creare solo un riquadro alla volta		In <i>landscape</i> possiamo creare un multi-riquadro
Non dipende dal frammento, ha un proprio ciclo di vita		Dipende dall'attività, non può esistere in maniera indipendente
Progetti difficili da gestire		Progetti meglio strutturati e gestione facilitata
Può usare 0 o più frammenti		Può essere riutilizzata in altre activity
Dobbiamo menzionarlo nel <i>Manifest</i>		Non è richiesta la dichiarazione nel <i>Manifest</i>
Usa molta memoria		Usa solo la memoria necessaria
Processo pesante		Processo leggero

A cosa servono i metodi `onSaveInstanceState()` e `onRestoreInstanceState()`?

- **`onSaveInstanceState()`**: viene chiamato prima di posizionare l'activity in stato di background, consentendo di salvare qualsiasi stato di istanza dinamico nella propria attività e nel pacchetto specificato, per essere successivamente ricevuto in `onCreate(Bundle)` se l'activity deve essere ricreata.
- **`onRestoreInstanceState()`**: ulteriore metodo per ripristinare lo stato dell'app, ma non è molto comune, viene chiamato dopo `onStart()`.

Si consideri il seguente frammento di codice usato per salvare lo stato dell'app prima di onDestroy():

```
public void onSaveInstanceState(Bundle savedInstanceState) {  
    savedInstanceState.putString("NOME", nome);  
    savedInstanceState.putInt("NUMERO", numero);  
    super.onSaveInstanceState(savedInstanceState);  
}
```

Si scriva il corrispondente frammento di codice, indicando anche in quale metodo deve essere inserito, per recuperare lo stato quando l'app viene ricreata.

Lo si recupera in onCreate():

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    if (savedInstanceState != null) {  
        nome = savedInstanceState.getString ("NOME");  
        numero = savedInstanceState.getInt("NUMERO");  
    }  
}
```

Si consideri la seguente situazione: un'app viene lanciata e l'utente interagisce con l'app; ad un certo punto l'utente ruota il dispositivo e continua ad interagire con l'app; infine l'utente chiude l'app tramite un apposito pulsante. In quali stati è passata l'app e quali metodi sono stati chiamati dal momento in cui l'app viene lanciata al momento in cui termina?

1. L'app viene lanciata e l'utente interagisce con l'app:

onCreate() -> Created -> onStart() -> Started -> onResume() -> Resumed;

2. L'utente ruota il dispositivo e continua ad interagire con l'app:

onPause() -> Paused -> onStop() -> Stopped -> onDestroy() -> Destroyed -> [punto 1]

3. L'utente chiude l'app tramite un apposito pulsante:

onPause() -> Paused -> onStop() -> Stopped -> onDestroy() -> Destroyed.

Si spieghi come un'app possa eseguire la seguente sequenza di metodi

- 1. **onCreate()**
- 2. **onStart()**
- 3. **onResume()**
- 4. **onPause()**
- 5. **onStop()**
- 6. **onRestart()**
- 7. **onStart()**
- 8. **onResume()**
- 9. **onPause()**
- 10. **onResume()**

L'utente avvia l'app ed interagisce con essa.

L'utente passa ad un'altra app, per poi riprenderla successivamente.

L'utente chiude l'app per poco tempo, e la riprende.

Un dispositivo Android può funzionare sia in modalità portrait (verticale) che landscape (orizzontale). Quando un dispositivo Android viene ruotato si passa dall'una all'altra modalità. Per gestire in maniera appropriata tali passaggi, di cosa si deve preoccupare il programmatore?

Quando l'utente ruota il dispositivo, l'activity viene prima eliminata eseguendo le callback: `onPause() → onStop() → onDestroy()`, e poi ricreata: `onCreate() → onStart() → onResume()`. Dunque si ha una perdita di stato.

Quando un dispositivo Android viene ruotato, passando da una modalità (portrait/landscape) all'altra (landscape/portrait), normalmente un'applicazione effettua un “cambio di configurazione”, pertanto necessita di esser prima distrutta per poi esser ricreata: questo fa sì che si perda lo stato dell'activity che l'utente utilizzava, se il programmatore non gestisce il cambiamento di stato.

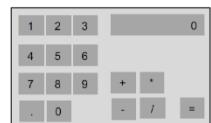
Per gestire tale cambiamento, di recente è stato implementato un attributo di `<application>`, `android:configChanges="orientation | keyboardHidden | screenSize"`: se lo si specifica, è possibile lasciare al sistema operativo la gestione, così che non venga perso alcun dato; tale attributo fa sì che venga invocato il listener `onConfigurationChanged()`, in cui è possibile eseguire qualsiasi computazione si desideri.

Un approccio programmatico che consente al programmatore la gestione manuale dei dati consiste nel salvare alcuni dati nell'oggetto `Bundle savedInstanceState`: quando c'è il cambiamento viene invocato il metodo `onSaveInstanceState(Bundle savedInstanceState)`, al cui interno vengono tipicamente aggiunte informazioni a tale oggetto (usando i setter `savedInstanceState.putString(String tag, ArrayList<String> list)`, `savedInstanceState.putInt(String tag, int value)`, ...) e, prima della chiusura del metodo, viene invocato `super.onSaveInstanceState(savedInstanceState)` - metodo di Activity - per rendere effettivo il salvataggio.

Questo oggetto viene passato anche al metodo `onCreate()`, per cui qui è possibile ricavare le informazioni sullo stato memorizzate attraverso i getter `savedInstanceState.getStringArrayList(String tag)`, `savedInstanceState.getInt(String tag)`, ...

Stai usando il tuo smartphone per fare dei conti con un app “calcolatrice”, come mostrato nella figura a sinistra. Involontariamente ruoti il dispositivo e ti ritrovi con la situazione descritta nella figura a destra. Il valore “783” è diventato “0” e la disposizione dei tasti è cambiata. Cosa è successo? Cosa ha fatto bene e cosa ha sbagliato il programmatore dell'app? Cosa avrebbe dovuto fare per ovviare all'errore?

Quando si ruota il dispositivo, l'app con cui l'utente sta interagendo, se non è stato ben gestito il cambio di configurazione, viene prima distrutta e poi riavviata (Quesito 3, punto 2), per cui (non essendo stato gestito il cambio) viene perso lo stato precedente.



La disposizione dei tasti è cambiata, giustamente, in quanto si è passati da portrait a landscape, per cui è necessario ridisegnare il layout: in questo caso, il programmatore ha gestito elegantemente il tutto.

Per non perdere lo stato precedente, di recente è stato implementato l'attributo `android:configChanges="orientation | screenSize | keyboardHidden"`, che consente all'app di non perdere lo stato precedente quando cambia l'orientamento del dispositivo, e permette di eseguire una qualsiasi computazione desiderata invocando il listener `onConfigurationChanged()`.

Un altro modo per tener traccia del valore calcolato (es. “783”) è utilizzare il listener `onSaveInstanceState(Bundle savedInstanceState)`, invocato prima dell'`onDestroy()`, che consente di aggiungere all'oggetto Bundle vari dati attraverso metodi `putInt(String tag, int value)`, `putFloat(String tag, float value)`, ...; infine, per conservare i cambiamenti occorre invocare

`super.onSaveInstanceState(savedInstanceState)`. Per riottenere i vari dati memorizzati nel Bundle, è possibile utilizzare l'oggetto Bundle passato all'`onCreate()`, invocandovi `getInt(String tag)`, `getFloat(String tag)`, ... ed inserire il valore nel `TextView` del risultato.

Così facendo, viene memorizzato lo stato precedente.

L'app TreSchermi consta di 3 activity, **Activity1**, **Activity2** e **Activity3**. Ognuna di queste activity prevede 3 pulsanti: uno per passare all'attività successiva (**Next**), uno per passare all'attività precedente (**Prev**) ed una per terminare l'app (**Stop**). Il pulsante **Next** dell'**Activity3** lancia l'**Activity1** ed il pulsante **Prev** dell'**Activity1** lancia l'**Activity3**. Si indichi la sequenza delle chiamate delle funzioni del ciclo di vita delle 3 attività nei seguenti due casi:

1. Avvio app (Activity1) – Pulsante Next – Pulsante Next – Pulsante Next – Pulsante Stop

2. Avvio app (Activity1) – Pulsante Prev – Pulsante Next – Pulsante Stop

CASO 1

1. Activity1.onCreate()
2. Activity1.onStart()
3. Activity1.onResume()
4. Activity1.onPause()

5. Activity2.onCreate()
6. Activity2.onStart()
7. Activity2.onResume()
8. Activity2.onPause()

9. Activity3.onCreate()
10. Activity3.onStart()
11. Activity3.onResume()
12. Activity3.onPause()

13. Activity1.onResume()

14. Activity1.onPause()
15. Activity1.onStop()
16. Activity1.onDestroy()
17. Activity2.onDestroy()
18. Activity2.onDestroy()
19. Activity3.onDestroy()
20. Activity3.onDestroy()

CASO 2

1. Activity1.onCreate()
2. Activity1.onStart()
3. Activity1.onResume()
4. Activity1.onPause()

5. Activity3.onCreate()
6. Activity3.onStart()
7. Activity3.onResume()

8. Activity3.onPause()
9. Activity1.onResume()

10. Activity1.onPause()
11. Activity1.onStop()
12. Activity1.onDestroy()
13. Activity3.onDestroy()
14. Activity3.onDestroy()

Nel contesto dell'esercizio precedente, si mostri un frammento di codice che potrebbe essere per l'`onClickListener` del pulsante **Next** dell'**Activity2**.

```
next.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent i = new Intent(this, Activity3.class);
        startActivity(i);
    }
});
```

INTENT

Cosa è un Intent? Si faccia un esempio in cui viene coinvolto un Intent utilizzando opportuni frammenti di codice.

Un intent è un semplice oggetto che viene utilizzato per la comunicazione tra componenti Android, come activity, content provider, receiver e broadcast transmission. Gli intent vengono anche utilizzati per trasferire dati tra attività. Gli intent vengono generalmente utilizzati per iniziare una nuova activity utilizzando `startActivity()`.

Si forniscano degli spezzoni di codice per il lancio di una nuova activity con un Intent esplicito e con un Intent implicito. Si spieghi cosa è necessario per lanciare l'Intent implicito.

Intent esplicito:

Un intento esplicito è quello che si usa per avviare un componente specifico dell'app, come una particolare activity o servizio nella app. Per creare un intento esplicito, bisogna definire il nome del componente nell'oggetto Intent: tutte le altre proprietà dell'intent sono facoltative.

Ad esempio, se si ha la necessità di dover lanciare una nuova Activity che svolge un'operazione si può fare nel seguente modo:

```
Intent intent = new Intent(getApplicationContext(), ActivityDaLanciare.class);
startActivity(intent); // oppure, startActivityForResult(intent, 0) per ricevere un risultato
```

Intent implicito:

Un intento implicito specifica un'azione che può invocare qualsiasi app sul dispositivo in grado di eseguire l'azione. L'uso di un intent implicito è utile quando l'app non è in grado di eseguire l'azione, ma altre app probabilmente possono e desideri che l'utente scelga quale app utilizzare. Gli intent impliciti vengono scelti dal sistema operativo in base a tre fattori: action, data e category; ognuno di essi ha un tag specifico nel manifesto, da inserire all'interno di un **<intent-filter>**. Bisogna implementare una category di default, altrimenti il sistema operativo non riconoscerà nessuno di quegli intent come ideale per risolvere la computazione necessaria.

```
Intent intent = new Intent(/*qui occorre specificare Action e/o Data e/o Category*/);
startActivity(intent); // oppure, startActivityForResult(intent, 0) per ricevere un risultato
```

Per lanciare l'intent implicito, come si nota dagli spezzoni di codice, non bisogna impostare manualmente la classe o la componente da lanciare, bensì occorre impostare (nel costruttore o attraverso i setter) i parametri Action, Data e Category: al lancio dell'activity, Android consulta il manifesto per cercare un'activity che soddisfi i parametri specificati.

Il metodo `startActivityForResult(Intent, int)` permette di lanciare una nuova activity ed alla sua terminazione di recuperare un valore di ritorno ed ulteriori dati inseriti in un intent attraverso il metodo `onActivityResult(int, int, Intent)`. Supponendo che l'intent non sia sufficiente né per passare i dati all'activity chiamata né per memorizzare i dati da restituire all'activity chiamante; per sfruttando i suddetti metodi, come risolveresti il problema?

Se un intent non è sufficiente né per passare i dati all'activity chiamata né per memorizzare i dati da restituire all'activity chiamante, è possibile sfruttare i parametri del metodo **`onActivityResult`**:

- **Int requestCode**, rappresenta il codice della richiesta passato al metodo **`startActivityForResult`**, ad esempio un numero pari a 0;
- **Int resultCode**, rappresenta il codice del risultato restituito dall'activity chiamata, col metodo **`setResult(int resultCode, Intent data)`**, ad esempio **RESULT_OK**;
- **Intent data**, rappresenta l'intent di ritorno avente i dati da restituire all'activity chiamante.

Una gestione accurata in **`onActivityResult`** di tali parametri prevede la verifica dei valori che essi possiedono: ad esempio, se non è stato possibile gestire la richiesta si può settare il **resultCode** ad un valore diverso da **RESULT_OK**, per cui in **`onActivityResult`** si procede con un semplice return nel caso in cui tale parametro è diverso da quello che ci si aspetta.

Si consideri il seguente snippet di codice dell'activity ActivityA:

```
Intent intent = new Intent(getApplicationContext(), ActivityB.class);
intent.putExtra("NOME", "Roberto");
intent.putExtra("VALORE", 32);
intent.putExtra("STATO", true);
startActivity(intent);
```

Si scriva lo snippet di codice per l'Activity B per recuperare i valori inseriti nell'intent.

```
Intent i = getIntent();
String nome = i.getStringExtra("NOME", null);
int valore = i.getIntExtra("VALORE", 0);
boolean stato = i.getBooleanExtra("STATO", false);
```

Mostrare un frammento di codice che permette ad un'activity di un'app di lanciare un'altra activity. Mostrare un altro frammento di codice in cui si lancia un'altra activity con l'intenzione di recuperare un valore di ritorno (nota: è richiesto solo il codice per lanciare l'activity, non quello per recuperare il risultato).

```
public void onClick(View v) {
    Intent i = new Intent();
    i.setClass(getApplicationContext(), Secondaria.class);
    startActivity(i);
}
```

```
public void onClick(View v) {
    Intent i = new Intent();
    i.setClass(getApplicationContext(), Secondaria.class);
    startActivityForResult(i, 0);
}
```

Si scrivano degli snippet di codice per lanciare da un activity “Principale” un’altra activity, “Secondaria”, passando un valore di tipo intero dall’activity principale a quella secondaria e facendo in modo che l’activity secondaria restituisca un valore di tipo stringa all’activity principale.

Attività Principale:

```
public void onClick(View v) {  
    Intent i = new Intent();  
    i.setClass(getApplicationContext(), Secondaria.class);  
    i.putExtra("INTERO", num);  
    startActivityForResult(i, 0);  
}
```

Attività Secondaria:

```
protected void setReturnIntent() {  
    Intent intent = getIntent();  
    String str = ...;  
    int totale = ...;  
  
    Intent data = new Intent();  
    data.putExtra("STRINGA", text);  
    setResult(RESULT_OK, data);  
}
```

Attività Principale:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    if(requestCode != 0) return;  
    if(resultCode != Activity.RESULT_OK) return;  
    if(data == null) return;  
    word = data.getStringExtra("STRINGA", null);  
    //  
}
```

Un’activity ActivityA ha bisogno di lanciare un’ActivityB passando a tale activity un intero x ed una stringa s. Cosa si deve fare?

Dettagliare la risposta con opportuni frammenti di codice.

In ActivityA.java, creiamo un intent impostando la classe ActivityB ed aggiungendo i valori voluti, per poi invocare il metodo startActivityForResult(Intent i) per lanciare l’activity senza ricevere un risultato:

```
Intent i = new Intent();  
i.setClass(getApplicationContext(), ActivityB.class);  
i.putIntExtra("intero", x); // si suppone che x sia stato definito in precedenza  
i.putStringExtra("stringa", s); // si suppone che s sia stata definita in precedenza  
startActivity(i);
```

In ActivityB.java, ricaviamo l’intent con getIntent(), i dati con getExtra(String tag) e procediamo con la computazione:

```
Intent i = getIntent();  
Integer x = getExtra("intero");  
String s = getExtra("stringa");  
...
```

THREAD

Android differenzia il thread principale dagli altri thread. Quali sono le operazioni non permesse nel thread principale (main) e quali quelle non permesse nei thread secondari (background)? Si motivi la risposta.

Le app Android vengono eseguite per impostazione predefinita sul thread principale, chiamato anche thread dell'interfaccia utente. Gestisce tutti gli input dell'utente e quelli di output, quindi si evitano operazioni che richiedono tempo, a differenza di un Thread secondario, che include chiamate di rete, decodifica di bitmap o lettura e scrittura dal database.

Un'app che utilizza una connessione Internet sfrutta la classe `AsyncTask`. Tale classe prevede i seguenti metodi: `onPreExecute()`, `doInBackground()`, `onProgressUpdate()`, `onPostExecute()`. Cosa viene tipicamente implementato in questi metodi? Nella risposta specificare per ogni metodo una possibile operazione da implementare nel metodo.

Quando viene eseguita un'attività asincrona, l'attività passa attraverso 4 passaggi:

- **`onPreExecute ()`**, richiamato sul thread dell'interfaccia utente prima dell'esecuzione dell'attività. Questo passaggio viene normalmente utilizzato per impostare l'attività, ad esempio mostrando una barra di avanzamento nell'interfaccia utente.
- **`doInBackground (Params ...)`**, richiamato sul thread in background immediatamente dopo che **`onPreExecute()`** termina l'esecuzione. Questo passaggio viene utilizzato per eseguire calcoli in background che possono richiedere molto tempo. I parametri dell'attività asincrona vengono passati in questo passaggio. Il risultato del calcolo deve essere restituito da questo passaggio. Questo passaggio può anche utilizzare **`publishingProgress (Progress ...)`** per pubblicare una o più unità di progresso. Questi valori sono pubblicati sul thread dell'interfaccia utente, nel passaggio **`onProgressUpdate (Progress ...)`**.
- **`onProgressUpdate (Progress ...)`**, richiamato sul thread dell'interfaccia utente dopo una chiamata a **`publishingProgress (Progress ...)`**. I tempi dell'esecuzione non sono definiti. Questo metodo viene utilizzato per visualizzare qualsiasi forma di progresso nell'interfaccia utente mentre il calcolo in background è ancora in esecuzione. Ad esempio, può essere utilizzato per animare una barra di avanzamento.
- **`onPostExecute (Result)`**, richiamato sul thread dell'interfaccia utente al termine del calcolo in background. Il risultato del calcolo in background viene passato a questo passaggio come parametro.

Per quali situazione occorre utilizzare la classe `AsyncTask`? Si faccia un esempio.

La classe `AsyncTask<Params, Progress, Result>` è una classe che consente di facilitare la gestione dei thread con Android nel caso in cui i thread in background devono comunicare con l'activity (UI) corrente.

I tipi generici che gestisce `AsyncTask` sono:

- **`Params`**, ossia il tipo dei parametri con cui il thread in background deve effettuare la computazione. Uno o più oggetti Params verranno passati al metodo **`doInBackground()`**;
- **`Progress`**, ossia il tipo dei parametri utilizzato per comunicare il progresso all'activity corrente. Nel metodo **`doInBackground()`**, ogni volta viene invocato **`publishProgress(Progress p)`**, viene invocato il listener sull'activity **`onProgressUpdate(Progress p)`**, per cui è possibile ora comunicare il progresso nell'interfaccia utente;
- **`Result`**, ossia il tipo dei parametri che si vuole venga restituito da **`doInBackground()`** e, di conseguenza, passato al listener **`onPostExecute(Result result)`**.

Ad esempio, si vuole caricare un'immagine utilizzando un thread. Viene creata una classe che estende `AsyncTask<Integer, Integer, Bitmap>`, dove: il primo Integer rappresenta l'id dell'immagine da caricare, il secondo Integer rappresenta il progresso da gestire con un valore intero, e Bitmap rappresenta il bitmap dell'immagine da ottenere.

Per un oggetto `AsyncTask`, a cosa serve il metodo `onProgressUpdate(Integer... progress)`? Viene eseguito nel thread creato per il task asincrono o nel main thread? Perché?

Il metodo **`onProgressUpdate (Progress ...)`** è richiamato sul thread dell'interfaccia utente dopo una chiamata a **`publishingProgress(Progress ...)`**. I tempi dell'esecuzione non sono definiti. Questo metodo viene utilizzato per visualizzare qualsiasi forma di progresso nell'interfaccia utente mentre il calcolo in background è ancora in esecuzione. Ad esempio, può essere utilizzato per animare una barra di avanzamento o mostrare i registri in un campo di testo.

Viene usato nel main Thread perché Android non permette ai thread in background di interagire con l'interfaccia utente, solo il main thread può farlo.

Si completi il seguente codice assumendo di avere a disposizione la funzione “partialLoad()” che si occupa di caricare in ogni chiamata un 10% dell’immagine img (quindi dopo dieci chiamate a tale funzione img sarà completa). Si renda visibile la ProgressBar all’inizio del caricamento e invisibile alla fine. Si aggiorni la progress bar ad ogni 10% di caricamento e si mostri un Toast di avviso “Caricamento quasi completato” quando si raggiunge l’80% del caricamento. Si mostri l’immagine nell’imageView alla fine del caricamento.

```
public class ThreadAsyncTaskActivity extends Activity {  
    private ImageView imageView;  
    private ProgressBar progressBar;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main_layout);  
        imageView = (ImageView) findViewById(R.id.imageView);  
        progressBar = (ProgressBar) findViewById(R.id.progressBar);  
    }  
    class LoadIconTask extends AsyncTask<Integer, Integer, Bitmap> {  
        private Integer index = 1;  
        @Override  
        protected void onPreExecute() {  
            progressBar.setVisibility(ProgressBar.VISIBLE);  
        }  
        @Override  
        protected Bitmap doInBackground(Integer... ids) {  
            Bitmap img = BitmapFactory.decodeResource(getResources(), ids[0]);  
            for(int i=1; i<=10; i++){  
                partialLoad();  
                publishProgress(i*10);  
            }  
            return img;  
        }  
        @Override  
        protected void onProgressUpdate(Integer... values) {  
            if(values[0] == 80) Toast.makeText(getApplicationContext(), "Caricamento quasi completato",  
            Toast.LENGTH_SHORT).show();  
            progressBar.setProgress(values[0]);  
        }  
        @Override  
        protected void onPostExecute(Bitmap result) {  
            progressBar.setVisibility(ProgressBar.INVISIBLE);  
            imageView.setImageBitmap(result);  
        }  
    }  
}
```

FRAGMENT

Si descriva il meccanismo dei Frammenti. Si faccia un esempio per illustrarne l'utilità.

I frammenti sono porzioni di interfaccia utente che hanno una vita propria nell'activity che li ospita: per questo, talvolta sono chiamati anche sub-activity. Essi hanno un differente ciclo di vita dall'activity, ma esso dipende da quello dell'activity che lo ospita: se l'activity ospitante viene distrutta, vengono distrutti anche i vari frammenti.

Un frammento può essere creato staticamente utilizzando un **tag <fragment>** da inserire nel layout dell'activity ospitante. Il layout del frammento, tuttavia, dev'essere specificato in un file XML separato. Nel momento in cui viene invocato il metodo **onCreateView()** del frammento, occorre effettuare **'linflate'** del file di layout corrispondente nel **tag <fragment>** del layout principale: dopo questa operazione, il frammento è **Created** nell'activity che lo ospita. Dopo che l'activity viene stoppata e prima che essa viene distrutta, viene effettuato il detatch del frammento (metodo **onDetach()**).

Una classe **Fragment** cura l'utilizzo dinamico del frammento nel corso dell'interazione con l'utente.

Ad esempio, un frammento può essere anche inserito dinamicamente nel layout a runtime. Occorre eseguire il codice posto di seguito:

```
FragmentManager fm = getFragmentManager();
FragmentTransaction ft = fm.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
ft.add(R.id.fragment_container, fragment); /* R.id.fragment_container è un ViewGroup nel layout dell'activity che individua la porzione dello schermo da dedicare a questo frammento */
```

Normalmente, un frammento non viene aggiunto nel backstack: si usa il metodo **addToBackStack()** per inserire i cambiamenti nel backstack; questo perché il backstack considera solo le activity, pertanto dobbiamo gestire manualmente i frammenti. Se non chiamiamo questo metodo, quando premiamo Back salteremo i cambiamenti fatti con i frammenti: non è quello che l'utente si aspetta.

Durante il corso è stato analizzato un esempio in cui occorreva far comunicare due frammenti. In tal senso, è bene specificare che un frammento dovrebbe evitare la comunicazione diretta con un altro frammento della stessa activity, in quanto ciò riduce la riusabilità; al contrario, è opportuno utilizzare l'activity ospitante come "comunicatore" tra il frammento mittente ed il frammento destinatario.

Per cosa sono utili i Frammenti?

Vengono usati i frammenti per costruire interfacce utenti dinamiche che permettono di adattarsi a diverse dimensioni degli schermi. Sono utili per encapsulare la logica dell'applicazione, per migliorare la gestione del ciclo di vita e per riutilizzarli in altre attività. Ad esempio mediante i frammenti si può decidere di inserire un **listView** che occupa metà dello schermo e un **webView** che occupa l'altra metà in modo tale che quando si clicca su un elemento del frammento A, viene passata l'informazione alla **webView** nel frammentoB e immediatamente si vede cosa viene ricercato senza dover effettuare un cambio di attività.

Se due frammenti di un activity devono comunicare è buona prassi di programmazione implementare tale comunicazione non in modo diretto da frammento a frammento ma passando attraverso l'activity che ospita i frammenti (quindi il frammento che vuole inviare la comunicazione lo fa interagendo con l'activity ospitante e poi questa interagisce con il frammento che deve ricevere la comunicazione). Perché è una buona prassi di programmazione? Si descriva un modo per implementare la comunicazione fra due frammenti attraverso l'activity ospitante.

È una buona prassi di programmazione perché favorisce il riuso del codice.

Si assuma di avere un'applicazione che, selezionato un nome di un autore, visualizzerà la sua citazione. Per quest'azione vengono utilizzati 2 frammenti (FragmentAutori, FragmentCitazioni). La prima operazione che deve essere effettuata è quella della visualizzazione dei nomi degli autori. Dunque nell'**onCreate()** della **MainActivity** istanzieremo un riferimento alla classe FragmentAutori mediante un **FragmentManager**. Per non far comunicare direttamente i 2 frammenti viene settato nella classe FragmentAutori, un metodo che setta in una variabile della classe FragmentAutori, un riferimento alla classe **MainActivity**, in maniera tale che, quando verrà selezionato un autore, la classe FragmentAutori, mediante un **onItemClick**, invocherà tramite il riferimento alla **MainActivity**, il metodo della classe **MainActivity** (**respond**) che gestirà la richiesta e invocherà il metodo del fragmentCitazione che mostrerà la citazione.

Scrivere il metodo **onCreate** di un'app che utilizza due frammenti e che all'avvio faccia partire i due frammenti. I due frammenti dividono lo schermo in due parti uguali. Mostrare anche i file xml per il layout. Si assume di avere a disposizione le classi **FrammentoA** e **FrammentoB** dei due frammenti.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    FragmentManager fm= getSupportFragmentManager();
    FragmentTransaction ft = fm.beginTransaction();

    FrammentoA fragmentA = new FrammentoA();
    FrammentoB fragmentB = new FrammentoB();

    ft.add(R.id.fragment_container, fragmentA);
    ft.add(R.id.fragment_container, fragmentB);
    ft.commit();
}
```

```
<FrameLayout
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="it.unisa.di.mp.myfragmentstatic.FrammentoA"
    android:layout_weight="1">

    //TO - DO

</FrameLayout>
```

Scrivere il frammento di codice da usare nel metodo onCreate() di un'activity che permette di inserire un solo frammento (Frammento1) se la larghezza del display è minore di 300dp e di inserire due frammenti (Frammento1 e Frammento2) se la larghezza del display è uguale o maggiore di 300dp. Si assume che il layout preveda un LinearLayout orizzontale con due frame per inserire i frammenti con id, rispettivamente, frame1 e frame2.

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    FragmentManager fm= getSupportFragmentManager();  
    FragmentTransaction ft = fm.beginTransaction();  
  
    Frammento1 fragmentA = new Frammento1();  
    Frammento2 fragmentB = new Frammento2();  
    ft.add(R.id.frame1, fragmentA);  
    float dpWidth = calcolaLarghezzaDisplayInDp();  
    if(dpWidth >= 300) ft.add(R.id.frame2, fragmentB);  
  
    ft.commit();  
}
```

State progettando un'app che utilizza dei frammenti. Avete già scritto la classe MioFrammento che implementa un singolo frammento. Adesso dovete scrivere nell'activity principale il codice che inserisce 4 frammenti MioFrammento. Il file di layout prevede 4 frame per ospitare i 4 frammenti e gli id dei 4 frame sono frameFrammento1, frameFrammento2, frameFrammento3, frameFrammento4. Mostrare il codice che serve a creare ed inserire dinamicamente i frammenti. Il costruttore della classe MioFrammento prende in input una stringa che servirà per mostrare il "nome" del frammento; pertanto dovete passare una stringa "Frammento i", dove i=1,2,3,4, per ognuno dei frammenti da creare ed inserire.

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    FragmentManager fm= getSupportFragmentManager();  
    FragmentTransaction ft= fm.beginTransaction();  
  
    MioFrammento mf1= new MioFrammento("MioFrammento1");  
    MioFrammento mf2= new MioFrammento("MioFrammento2");  
    MioFrammento mf3= new MioFrammento("MioFrammento3");  
    MioFrammento mf4= new MioFrammento("MioFrammento4");  
  
    ft.add(R.id.frameFrammento1, mf1);  
    ft.add(R.id.frameFrammento2, mf2);  
    ft.add(R.id.frameFrammento3, mf3);  
    ft.add(R.id.frameFrammento4, mf4);  
  
    ft.commit();  
}
```

NETWORKING

Quando un'app ha bisogno di comunicare via Internet che cosa è necessario fare? Si faccia un esempio usando uno snippet di codice (senza dettagli ma solo con la struttura generale).

Per eseguire operazioni di rete nell'applicazione, il manifest deve includere le seguenti autorizzazioni:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Per evitare di creare un'interfaccia utente che non risponde, non eseguire operazioni di rete sul thread dell'interfaccia utente. Per impostazione predefinita, Android 3.0 (livello API 11) e versioni successive richiede di eseguire operazioni di rete su un thread diverso dal thread dell'interfaccia utente principale; se non lo si fa, viene lanciata una **NetworkOnMainThreadException**.

Il frammento Activity seguente utilizza un headless *Fragment* per incapsulare operazioni di rete asincrone.

Il seguente abbozzo di classe NetworkTask mostrato viene utilizzato per ricevere da Internet dei dati nella forma di Stringa. Nella parte di ricezione (non dettagliata) viene eseguito un ciclo per 100 iterazioni e ad ogni iterazione viene ricevuta una stringa str mentre la stringa received_data viene utilizzata per memorizzare tutti i dati ricevuti che alla fine verranno visualizzati nel TextView textResponse. Nel layout c'è una progressBar utilizzata per mostrare in quale iterazione ci troviamo. Questo codice non funziona. Perché? Come possiamo risolvere il problema?

```
class NetworkTask extends AsyncTask<Integer, Integer, String> {
    @Override
    protected void onPreExecute() {
        progressBarDownload.setVisibility(ProgressBar.VISIBLE);
    }
    @Override
    protected String doInBackground(Integer...values) {
        //Do something
        String received_data = "";
        for (int i=0; i<100; i++) {
            progressBarDownload.setProgress(i);
            //Ricevi i dati nella stringa str
            received_data += str;
        }
        return received_data;
    }
    @Override
    protected void onPostExecute(String data) {
        progressBarDownload.setVisibility(ProgressBar.INVISIBLE);
        textResponse.setText(data);
    }
}
```

Il codice sopra scritto non funziona in quanto, quando il metodo doInBackground() è in esecuzione, non si può interagire con l'activity principale (dato che ci si trova in un thread separato), per cui non è possibile impostare il progresso in tale metodo (dovrebbe esser lanciato un errore). Per risolvere, in doInBackground() va eliminata la prima riga di codice all'interno del ciclo for, sostituendola con publishProgress(i): questo metodo invocherà onProgressUpdate(Integer i) sull'activity principale, per cui al suo interno andremo ad impostare il progresso della progressBar.

Inoltre, se il ciclo for passa ad i valori da 0 a 99, la progressBar non verrebbe mai riempita del tutto (non è un errore, ma è bene precisare). Infine, in onPostExecute() andrebbe settata la progress bar a 0 (non è un errore, ma è bene precisare).

STORAGE

In che modo (o modi) varie activity che fanno parte della stessa app possono condividere dati? Si discuta dei vantaggi e svantaggi di ciascuno dei modi descritti.

Varie activity facenti parte della stessa app possono condividere dati in tre modi: attraverso **SharedPreferences**, files o basi di dati.

- **SharedPreferences** sono delle preferenze (localizzate in opportuni file di minime dimensioni) condivise tra più activity, che consentono la memorizzazione, tipicamente, di valori primitivi (interi, caratteri, booleani, ...) attraverso setter e getter. Per ricavare le preferenze condivise, è possibile utilizzare il metodo **getSharedPreferences(Context c)** oppure **getSharedPreferences(String filename)**. Per salvare informazioni, occorre un oggetto **SharedPreferences.Editor**, ricavabile attraverso il metodo **edit()** sulle preferenze ricavate: su quest'oggetto è possibile invocare i vari setter per salvare dati primitivi; dopo aver inserito i vari dati, occorre effettuare il **commit()** dell'editor.
- **File** sono il meccanismo più semplice ed efficiente per salvare informazioni (anche dati non primitivi, cioè oggetti serializzabili). I file utilizzabili con Android possono risiedere in tre memorie differenti: interna, esterna e cache; chiaramente, le directory per utilizzare i file nelle tre categorie precedenti sono diverse, ed esistono metodi che consentono di ricavarli in modo semplice. Il meccanismo di lettura/scrittura su file è, ovviamente, quello classico di Java. Per scrivere su file sulla memoria esterna è necessario richiedere l'opportuno permesso. Ogni applicazione in Android ha la sua directory privata, a cui possono accedere le sole activity della stessa applicazione.
- **Database** sono delle strutture che consentono la memorizzazione persistente di informazioni. Android consente un utilizzo semplificato (seppur analogamente complesso da utilizzare) di basi di dati relazionali SQL attraverso **SQLite**.

Vantaggi e svantaggi di ciascuno dei modi descritti dipendono principalmente da tre fattori: dati memorizzabili, semplicità ed efficienza. Le **SharedPreferences** possono memorizzare soltanto dati primitivi, sono molto semplici da utilizzare ed efficienti, in quanto trattano dati atomici facilmente ricavabili e memorizzabili in file opportuni di basse dimensioni. I file possono memorizzare qualsiasi oggetto al loro interno (nel caso in cui siano file puramente testuali o binari), sono semplici ed efficienti da utilizzare. I database sono strutture che consentono di memorizzare dati strutturati e relazionati in tabelle (quindi, si parla di dati complessi) persistentemente, sono (non molto) semplici da utilizzare, ma sono spesso inefficienti.

Si descrivano le varie possibilità offerte da Android per la memorizzazione statica (file) dei dati.

Per ogni app il sistema operativo prevede una directory privata, solo l'app può accedere a questa directory, all'interno della quale viene scritto un file per la memorizzazione dei dati.

Per creare e scrivere un **file**:

1. Chiamare **openFileOutput(fileName, mode)**, restituisce un **OutputStream**
2. Scrivere nel file (**write()**)
3. Chiudere lo stream (**close()**)

La modalità di accesso può essere:

- **MODE_PRIVATE** (file accessibile solo all'app)
- **MODE_APPEND**

Per leggere un file

1. Chiamare **openFileInput(fileName)**, restituisce un **InputStream**
2. Leggere dal file (**read()**)
3. Chiudere lo stream (**close()**)

Per i **file temporanei** si può usare una directory cache, Android cancellerà i file in questa directory SE necessario (manca spazio).

Android permette l'utilizzo di una **memoria esterna**, tipicamente una SD card. La memoria esterna può essere rimossa, quindi non si può assumere che i file siano sempre disponibili.

Si descrivano le SharedPreferences. Cosa sono? Come funzionano? Che differenza c'è fra SharedPreferences e Preferences?

Le **SharedPreferences** sono uno dei tre meccanismi utilizzati per effettuare in modo persistente la memorizzazione di dati condivisi tra più activity della stessa applicazione in un file di piccole dimensioni. Le preferenze condivise permettono la memorizzazione di soli dati primitivi (interi, caratteri, floating points, booleani, ...); tuttavia, sono molto efficienti e facili da usare.

Per ottenere le preferenze condivise, è possibile usare uno dei seguenti metodi:

- **getSharedPreferences()**, che ricava le preferenze condivise di default dell'applicazione;
- **getSharedPreferences(String filename)**, che ricava le preferenze condivise nel file specificato.

Per inserire un valore nelle **SharedPreferences**, occorre dapprima ricavare uno **SharedPreferences.Editor** attraverso il metodo **edit()** sull'oggetto **SharedPreferences** ricavato; in seguito, è possibile chiamare uno dei metodi **putInt(String tag, int value)**, **putChar(String tag, char value)**, ..., sull'editor; dopo aver inserito i dati, occorre effettuare il **commit()** dell'editor.

Per ricavare un valore dalle **SharedPreferences**, semplicemente s'invoca uno dei metodi seguenti sull'oggetto **SharedPreferences** ricavato: **getInt(String tag)**, **getChar(String tag)**, ...

Bisogna fare attenzione quando si invoca il metodo **getSharedPreferences()**. Esiste, infatti, un metodo **getPreferences()** che ricava le preferenze della singola activity, pertanto NON CONDIVISE tra tutte le activity dell'applicazione, a differenza di **getSharedPreferences()**.

Si consideri il salvataggio di dati tramite le SharedPreferences. Si forniscano gli opportuni frammenti di codice per memorizzare una stringa (String), un intero (Int) ed un intero lungo (Long) e per recuperarli. Si indichi anche dove i frammenti di codice vanno inseriti.
Per effettuare la memorizzazione permanente di dati primitivi condivisi da varie activity, utilizzare le **SharedPreferences** rappresenta una modalità molto semplice.

Solitamente, la memorizzazione avviene prima della distruzione dell'activity, per cui:

- in **onDestroy()**, per memorizzarli:

```
SharedPreferences prefs = getDefaultSharedPreferences();
SharedPreferences.Editor editor = prefs.edit();
editor.putString("stringa", string); // si assume che string sia stata definita in precedenza
editor.putInt("intero", integer); // si assume che integer sia stato definito in precedenza
editor.putLongInt("longint", l_int);
editor.commit()
```

- in **onCreate()**, per recuperarli:

```
SharedPreferences prefs = getDefaultSharedPreferences();
String string = prefs.getString("stringa");
int integer = prefs.getInt("intero");
long l_int = prefs.getLongInt("longint");
```

A cosa servono le classi DatabaseOpenHelper?

Per la gestione di tutte le operazioni relative al database, è stata data una classe di supporto che si chiama **SQLiteOpenHelper**. Gestisce automaticamente la creazione e l'aggiornamento del database. La sua sintassi è riportata di seguito:

```
public class DBHelper extends SQLiteOpenHelper {
    public DBHelper(){
        super(context,DATABASE_NAME,null,1);
    }
    public void onCreate(SQLiteDatabase db){}
    public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion){}
}
```

Si spieghi il ruolo della classe Cursor per l'utilizzo dei database SQL.

Contiene il set di risultati di una query effettuata su un database in Android. La classe Cursor ha un'API che consente a un'app di leggere (in modo sicuro dal tipo) le colonne restituite dalla query e di scorrere le righe del set di risultati.

Una volta che un cursore è stato restituito da una query del database, un'app deve scorrere il set di risultati e leggere i dati della colonna dal cursore. Internamente, il cursore memorizza le righe di dati restituiti dalla query insieme a una posizione che punta alla riga di dati corrente nel set di risultati. Quando un cursore viene restituito da un metodo `query()`, la sua posizione punta allo spot prima della prima riga di dati. Ciò significa che prima di poter leggere qualsiasi riga di dati dal cursore, è necessario spostare la posizione in modo che punti a una riga di dati valida. Ci sono vari metodi per la manipolazione della posizione del cursore.

```
Cursor resultSet = mydatabase.rawQuery("Select * from TutorialsPoint",null);
resultSet.moveToFirst();
String username = resultSet.getString(0);
String password = resultSet.getString(1);
```

Che cosa fa il seguente metodo?

```
private Cursor readSelectedEntries() {
    String[] projection = {
        SchemaDB.Tavola._ID,
        SchemaDB.Tavola.COLUMN_NAME,
        SchemaDB.Tavola.COLUMN_VOTO
    };
    String sortOrder = SchemaDB.Tavola.COLUMN_VOTO + " ASC";
    String selection = SchemaDB.Tavola.COLUMN_NAME + " LIKE ? + " and " + SchemaDB.Tavola.COLUMN_VOTO + " > ? ";
    String[] selectionArgs = {"Car%", "25"};
    Cursor cursor = db.query(
        SchemaDB.Tavola.TABLE_NAME,
        projection,
        selection,
        selectionArgs,
        null,
        null,
        sortOrder
    );
    return cursor;
}
```

ANIMAZIONI

Come funziona il meccanismo di layout e la relativa fase di misurazione delle view?

Il meccanismo di layout si compone di 3 fasi:



Cosa è l'albero delle view? Quali sono e come funzionano le due fasi necessarie per la visualizzazione del layout?

L'albero delle view è l'insieme dei widget che formano un determinato layout, strutturato secondo una gerarchia ad albero in quanto ogni file XML ne conserva la struttura. Alla radice dell'albero si trova un singolo ViewGroup, al livello successivo sono posti i suoi figli, al livello successivo sono posti i suoi "nipoti" e così via, fin quando non si raggiungono le foglie dell'albero.

Le fasi necessarie per la visualizzazione del layout sono tre:

- 1) **Misurazione**: fase in cui viene fatta una visita top-down dell'albero, per cui si misura width ed height della radice, e si esegue la stessa misurazione per ogni sottoalbero contenuto nell'albero delle view, ricorsivamente. In questa fase si effettuano due misurazioni: una misurazione "temporanea" che indica quanto grande, ogni view, vorrebbe essere, e una misurazione che indica quanto grande ogni view sarà realmente;
- 2) **Posizionamento**: fase in cui viene fatta una visita top-down dell'albero, per cui si posiziona dapprima la radice, ed è proprio la radice a posizionare ognuno dei suoi figli, così come ogni figlio posizionerà ognuno dei suoi figli, e così via fin quando non si raggiungono le foglie dell'albero;
- 3) **Disegno**: fase in cui si disegna ogni view nel layout dell'interfaccia.

Si spieghi come avviene la misurazione e il posizionamento delle view di un layout. Perché in alcuni casi i metodi `v.getWidth` e `v.getHeight`, dove `v` è una view del layout, usati in `onCreate()` restituiscono 0?

Le view di un layout, prima che vengano effettivamente disegnate nell'interfaccia utente attraverso la fase di draw, devono essere prima misurate, poi posizionate correttamente nell'UI. Bisogna prima specificare che ogni layout possiede il cosiddetto albero delle view (siccome ogni file XML è strutturato gerarchicamente in un albero).

La fase di **misurazione** ricava la misura che le varie view richiedono e calcola la misura reale che le varie view avranno, in relazione alla view genitore e ad i suoi fratelli (cioè, le view che risiedono nello stesso livello dell'albero): si parte dalla radice e si analizza ognuno dei sottoalberi di ogni figlio ricorsivamente, fin quando tutte le view sono state misurate.

La fase di **posizionamento** prevede il posizionamento delle view del layout secondo le misure calcolate nella fase precedente.

In alcuni casi, nel metodo `onCreate()` non è ancora possibile ricavare le misure (width ed height) delle varie view, in quanto è possibile che esse non siano state ancora posizionate e disegnate nel layout, anche dopo aver invocato il metodo `setContentView()`.

Che cosa è e a cosa serve l'oggetto **Canvas**? Si illustri, con sufficienti dettagli, un esempio in cui è necessario utilizzare un oggetto **Canvas**.

L'oggetto **Canvas** consente di disegnare una view customizzata in un determinato punto del layout.

Lo step più importante per disegnare una view personalizzata è eseguire l'override del metodo **onDraw()**, che riceve come parametro un oggetto **Canvas** che la view può usare per disegnare sé stessa. La classe **Canvas** definisce metodi per disegnare testo, linee, bitmap e molte altre grafiche primitive. Si possono usare i suoi metodi in **onDraw()** per creare la propria UI customizzata.

Prima di poter chiamare qualsiasi metodo di disegno, tuttavia, è necessario creare uno o più oggetti **Paint**.

Ad esempio, un **Canvas** fornisce un metodo per disegnare una linea, mentre **Paint** fornisce metodi per definire il colore della linea; **Canvas** fornisce un metodo per disegnare un rettangolo, mentre **Paint** definisce se riempire tale rettangolo con un colore o lasciarlo vuoto. Semplicemente, **Canvas** definisce le forme che si possono disegnare sullo schermo, mentre **Paint** definisce colore, stile, font e così via, per ogni forma si vuole disegnare.

Durante il corso è stato analizzato un esempio in cui si disegnava un Pentagramma su cui era possibile aggiungere varie note.

Si spieghi come si creano le animazioni grafiche (magari descrivendo tutte le possibilità) in Android possibilmente facendo un esempio concreto (magari usando solo una o due delle varie possibilità descritte).

Android permette di definire delle animazioni da applicare alle immagini, descritte con file XML:

- Rotazione
- Traslazione
- Scaling
- Trasparenza

Con controlli di vario tipo usufruendo dei parametri, come pivot, velocità, etc....

Come funzionano le animazioni? Come si può animare un oggetto grafico in Android? Si faccia un esempio scegliendo una specifica animazione.

Le animazioni consentono ad una determinata view di aggiungervi degli effetti speciali del tipo, tra gli altri: rotazione, traslazione, scaling, trasparenza. Per creare un'animazione, occorre creare un file di layout XML: alla radice dell'albero dev'essere posto un elemento `<set>` (oppure `<objectAnimator>` e `<valueAnimator>`), che indica l'insieme di animazioni da eseguire, al cui interno è possibile porre `<animator>` ed `<objectAnimator>`, ossia le singole animazioni, o ulteriori elementi `<set>`.

Ogni file d'animazione può essere memorizzato in un oggetto Animation. Per assegnare un'animazione ad un oggetto animation, occorre invocare il metodo statico `AnimationUtils.loadAnimation(Context context, int resource)`, che ritorna un oggetto Animation.

Un'animazione può essere utilizzata sulla maggior parte delle View di un layout, utilizzando il metodo `View.startAnimation(Animation animation)`.

```
<rotate
    android:startOffset="0"
    android:duration="4000"
    android:fromDegrees="0"
    android:toDegrees="180"
    android:pivotX="50%"
    android:pivotY="50%"
    />
```

Nella nostra MainActivity verrà istanziato un oggetto Animation che verrà egualato nel seguente modo:

```
AnimationUtils.loadAnimation(getApplicationContext(), R.anim.rotazione);
```

Verrà definito un metodo invocato al click del bottone che starterà l'oggetto con la relativa animazione.

Si scriva un file XML con un'animazione che permette di traslare prima un oggetto prima in orizzontale per 500 pixels in 1 secondo e poi in verticale per 300 pixel in 2 secondi. Normalmente dopo l'animazione l'oggetto ritorna automaticamente nella posizione iniziale; cosa si può fare per farle rimanere nella posizione finale dell'animazione?

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:shareInterpolator="false"

    <translate
        android:startOffset="0"
        android:duration="1000"
        android:fromXDelta="0"
        android:fromYDelta="0"
        android:toXDelta="500"
        android:toYDelta="0" />

    <translate
        android:startOffset="1000"
        android:duration="2000"
        android:fromXDelta="0"
        android:fromYDelta="0"
        android:toXDelta="0"
        android:toYDelta="300" />

    </set>
```

Per far sì che un oggetto traslato non ritorni nella sua pozione iniziale dopo l'animazione, è possibile procedere come segue:

1. Aggiungere un ***AnimationListener*** all'animazione che effettua la traslazione:

```
animation.setAnimationListener(new Animation.AnimationListener(){
    @Override
    public void onAnimationStart(Animation animation){ }
    @Override
    public void onAnimationEnd(Animation animation){ }
    @Override
    public void onAnimationRepeat(Animation animation){ }
});
```

2. Nel metodo ***onAnimationEnd()***, chiamare il metodo `public void layout(int left, int top, int right, int bottom)` sulla view traslata, ed aggiornare i parametri opportuni. Un altro modo è settare i parametri di layout ricavando i ***LayoutParams***, settandoli ed aggiungerli alla View.

Il seguente snippet di codice esegue prima un'animazione e poi rimuove l'oggetto dalla view parent (gli oggetti `image`, `animation` e `parentView` sono stati in precedenza opportunamente inizializzati):

```
...
image.startAnimation(animation);
parentView.removeViewAt(0);
...
```

Tuttavia l'effetto è quello di rimuovere immediatamente l'immagine senza dare il tempo all'animazione di essere eseguita. Perché accade ciò? Come si può ovviare al problema?

Il problema precedentemente descritto accade in quanto il metodo `image.startAnimation()` non interrompe l'esecuzione del codice, e la duration dell'animazione viene sorvolata dall'esecuzione del programma. Per ovviare al problema, si potrebbe aggiungere un `AnimationListener` all'animazione, ed al termine (eseguendo l'override del metodo listener `onAnimationEnd()`) rimuovere l'oggetto dalla view `parent`. Un altro modo potrebbe essere l'implementazione di un `AsyncTask<ImageView, Integer, Boolean>`, in modo tale che in `doInBackground(ImageView imageView)` venga avviata l'animazione e ritornato true, e in `onPostExecute(Boolean boolean)` venga rimosso l'oggetto se boolean == true.

La seguente classe implementa una view customizzata per un pentagramma. Così come mostrato l'oggetto Pentagramma viene disegnato con una nota in una specifica posizione (alle coordinate 350,78). Descrive le modifiche da fare per fare in modo che inizialmente il pentagramma sia vuoto e che successivamente si possano aggiungere delle note al pentagramma.

```
public class Pentagramma extends View {
    int MARGIN_TOP= 50;
    int MARGIN_H= 40;
    int LINE_SPACING = 8;
    int NOTEHEAD_SIZE = 6;
    int STEM_LENGTH = 30;
    public class Nota { ... }
    ArrayList<Nota> notesList = new ArrayList<Nota>();
    public Pentagramma(Context c, int screen_w, int screen_h) {
        super(c);
        setMinimumWidth(screen_w);
        setMinimumHeight(150);
        Nota nota = new Nota(350, 78);
        notesList.add(nota);
    }
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) { ... }
    @Override
    protected void onLayout(boolean b, int x1, int y1, int x2, int y2) { ... }
    @Override
    protected void onDraw(Canvas canvas) {
        Paint paint = new Paint();
        paint.setColor(Color.BLACK);
        int h = MARGIN_TOP;
        int x_start = MARGIN_H;
        int x_end = canvas.getWidth() - MARGIN_H;
        for (int i = 0; i<5; i++) {
            canvas.drawLine(x_start,h,x_end,h,paint);
            h = h + LINE_SPACING;
        }
        Nota n = notesList.get(0);
        int note_position_horizontal = n.getH();
        int note_position_vertical = n.getV();
        canvas.drawCircle(note_position_horizontal, note_position_vertical, NOTEHEAD_SIZE, paint);
        canvas.drawLine(note_position_horizontal+5, note_position_vertical-STEM_LENGTH,
                      note_position_horizontal+5, note_position_vertical, paint);
    }
}
```

Prima di tutto, occorre eliminare le ultime cinque righe del codice, ed eliminare le ultime due righe del costruttore del Pentagramma. Per inserire note in un pentagramma vuoto, è necessario implementare o un onclick listener o un `ontouch listener` sulla view del pentagramma: è preferibile utilizzare il secondo, in quanto il `MotionEvent` consentirà di ricavare il valore di asse x e y del punto in cui è stato effettuato il tocco. Dunque, Pentagramma implementerà l'interfaccia del `listener ontouch (OnTouchListener)`, per qui definirà il metodo `onTouch`: al suo interno ricaviamo X ed Y del punto in cui è stato effettuato il tocco, istanziamo una nuova nota passando i due valori precedenti, aggiungiamo la nota all'`ArrayList<Nota>` `notesList` e terminiamo con un `invalidate()` del pentagramma: questo fa sì che il pentagramma venga ridisegnato, per cui all'interno di `onDraw(Canvas canvas)` inseriamo un `for (int i=0; i<notes.size(); i++)` al cui interno si disegnano tutte le note inserite fino a quel momento (potrebbe risultare inefficiente: si potrebbe, per questo motivo, effettuare il disegno della singola nota effettuando gli accorgimenti opportuni).

Si fornisca un file XML per un'animazione che prima ruota la view di 180° e poi la sposta orizzontalmente di 200dp.

```
<set xmlns:android="http://schemas.android.com/apk/res/android">  
    android:shareInterpolator="false" >  
        <rotate  
            android:startOffset="0"  
            android:duration="xxxx"  
            android:fromDegrees="0"  
            android:toDegrees="180"  
            ... />  
        <translate  
            android:startOffset="xxxx"  
            android:duration="yyyy"  
            android:fromXDelta="0"  
            android:fromYDelta="0"  
            android:toXDelta="200"  
            android:toYDelta="0" />  
</set>
```

Si scriva un file xml per la seguente animazione di un oggetto drawable:

1. rotazione di 2 giri completi a destra, dal tempo 0 al tempo 2 sec
2. traslazione di 300px a destra, dal tempo 3 sec al tempo 4 sec,
3. rotazione di 2 giri completi a sinistra, dal tempo 3 sec al tempo 5 sec.

```
<rotate  
    android:startOffset="0"  
    android:duration="2000"  
    android:fromDegrees="0"  
    android:toDegrees="720"  
    />  
<translate  
    android:startOffset="3000"  
    android:duration="1000"  
    android:fromXDelta="0"  
    android:fromYDelta="0"  
    android:toXDelta="300"  
    android:toYDelta="0"  
    />  
<rotate  
    android:startOffset="3000"  
    android:duration="2000"  
    android:fromDegrees="0" (720)  
    android:toDegrees="-720" (0)  
    />
```

Si scriva un file animation.xml che permette di implementare la seguente animazione di una view: spostamento orizzontale di 300 px verso destra, seguito da una rotazione di 2 giri completi in senso orario con pivot al centro dell'oggetto, a sua volta seguita da un nuovo spostamento in orizzontale di 300px verso sinistra che riporta l'oggetto nella posizione originaria.

<pre><translate android:startOffset="0" android:duration="2000" android:fromXDelta="0" android:fromYDelta="0" android:toXDelta="300" android:toYDelta="0" /> <rotate android:startOffset="2000" android:duration="5000" android:fromDegrees="0" android:toDegrees="720" android:pivotX= "80%" android:pivotY= "50%" /></pre>	<pre><translate android:startOffset="7000" android:duration="2000" android:fromXDelta="0" android:fromYDelta="0" android:toXDelta="-300" android:toYDelta="0" /></pre>
--	--

MULTITOUCH

Che cosa è un oggetto **MotionEvent**? Si forniscano dettagli sul suo utilizzo.

Un **MotionEvent** è un oggetto utilizzato per segnalare eventi di movimento (mouse, penna, dito), rappresenta un singolo pointer e a volte più pointer.

Ad esempio, quando l'utente tocca per la prima volta lo schermo, il sistema invia un evento di tocco all'appropriato **View** con il codice di azione **ACTION_DOWN** e un insieme di valori degli assi che includono le **coordinate X** e **Y** del tocco e informazioni sulla pressione, le dimensioni e l'orientamento di l'area di contatto.

La classe **MotionEvent** fornisce molti metodi per interrogare la posizione e altre proprietà di puntatori, come **getX(int)**, **getY(int)**, **getAxisValue(int)**, **getPointerId(int)**, **getToolType(int)**, e molti altri. La maggior parte di questi metodi accetta l'indice del puntatore come parametro anziché l'id del puntatore. L'indice del puntatore di ciascun puntatore nell'evento varia da 0 a uno in meno del valore restituito da **getPointerCount()**.

Per ottenere un'azione di un **MotionEvent** si dovrebbe sempre usare il metodo **getActionMasked()**, è progettato per funzionare con più puntatori. Restituisce l'azione mascherata che viene eseguita, senza includere i bit dell'indice del puntatore.

Si descriva il meccanismo del Multitouch. Come vengono rappresentati i movimenti? Che cosa è un “pointer”? Che cosa è un MotionEvent?

Il Multitouch è un meccanismo che permette al dispositivo di reagire opportunamente in base al numero di dita che toccano lo schermo. Il movimento è rappresentato con: **ACTION_CODE** (cambiamento avvenuto) e **ACTION_VALUES** (posizione e proprietà del movimento). Ogni evento è rappresentato da un “pointer”, mentre **MotionEvent** rappresenta uno o più pointer. Alcuni eventi sono:

- **ACTION_DOWN**, che rappresenta la pressione di un singolo dito;
- **ACTION_POINTER_DOWN**, che rappresenta la pressione di un secondo, terzo, ..., dito;
- **ACTION_MOVE**, che rappresenta il movimento di uno o più dita;
- **ACTION_POINTER_UP**, che rappresenta il rilascio di un secondo, terzo, ..., dito dallo schermo;
- **ACTION_UP**, che rappresenta il rilascio dell'ultimo dito dallo schermo.

Ogni pointer ha un indice: se un singolo dito effettua la pressione, esso ha indice 0; se un secondo dito effettua la pressione, il primo avrà indice 0 ed il secondo avrà indice 1; e così via. Questi indici vengono memorizzati per individuare quale pointer ha eseguito una determinata operazione. Ogni pointer ha un id univoco per tutta la durata della pressione, fino al momento dopo il rilascio.

MotionEvent è la classe che cura la gestione degli eventi elencati in precedenza: essa rappresenta un movimento registrato da una periferica (mouse, trackball, penna, dita, ...). È possibile invocare numerosi metodi su un oggetto **MotionEvent**, quali:

- **getActionMasked()**, che ritorna l'action code catturato;
- **getActionIndex()**, che ritorna l'indice del pointer che ha scaturito l'evento;
- **getPointerId(int pointerIndex)**, che ritorna l'ID del pointer specificato;

Il seguente frammento di codice mostra un OnTouchListener per un MotionEvent. Si completi il codice facendo in modo che la variabile counter (si assuma che tale variabile sia accessibile globalmente) contenga sempre il numero di dita che stanno toccando lo schermo.

```
int counter;
public boolean onTouch(View v, MotionEvent event) {
    switch(event.getActionMasked()) {
        case MotionEvent.ACTION_DOWN:
            counter++; // potrebbe anche essere counter=1, in quanto è il primo dito che effettua pressione
            break;
        case MotionEvent.ACTION_POINTER_DOWN:
            counter++; // [?] potrebbe anche essere event.getActionIndex()+1
            break;
        case MotionEvent.ACTION_MOVE:
            break;
        case MotionEvent.ACTION_POINTER_UP:
            counter--; // [?] potrebbe anche essere event.getActionIndex()+1
            break;
        case MotionEvent.ACTION_UP:
            counter--; // potrebbe anche essere counter=0, in quanto è l'ultimo dito che rilascia la pressione
    }
    return true;
}
```

L'oggetto `MotionEvent` viene usato per descrivere i tocchi dell'utente sullo schermo. I tocchi ed i movimenti vengono codificati attraverso dei codici (riportati sotto) e dei "pointer ID".

- `MotionEvent.ACTION_DOWN`
- `MotionEvent.ACTION_POINTER_DOWN`
- `MotionEvent.ACTION_MOVE`
- `MotionEvent.ACTION_POINTER_UP`
- `MotionEvent.ACTION_UP`

Quale sequenza di questi codici-pointer ID viene generata nei seguenti casi?

1. Pollice tocca lo schermo, indice tocca lo schermo, pollice e indice si muovono, pollice lascia lo schermo, indice lascia lo schermo

Pollice tocca lo schermo	<code>MotionEvent.ACTION_DOWN</code>	0
Indice tocca lo schermo	<code>MotionEvent.ACTION_POINTER_DOWN</code>	1
Pollice e indice si muovono	<code>MotionEvent.ACTION_MOVE</code>	0,1
Pollice lascia lo schermo	<code>MotionEvent.ACTION_POINTER_UP</code>	0
Indice lascia lo schermo	<code>MotionEvent.ACTION_UP</code>	1

2. Pollice tocca lo schermo, medio tocca lo schermo, mignolo tocca lo schermo, medio lascia lo schermo, pollice lascia lo schermo, mignolo lascia lo schermo

Pollice tocca lo schermo	<code>MotionEvent.ACTION_DOWN</code>	0
Medio tocca lo schermo	<code>MotionEvent.ACTION_POINTER_DOWN</code>	1
Mignolo tocca lo schermo	<code>MotionEvent.ACTION_POINTER_DOWN</code>	2
Medio lascia lo schermo	<code>MotionEvent.ACTION_POINTER_UP</code>	1
Pollice lascia lo schermo	<code>MotionEvent.ACTION_POINTER_UP</code>	0
Mignolo lascia lo schermo	<code>MotionEvent.ACTION_UP</code>	2

State sviluppando un'app che permette di sfruttare il tocco multiplo sullo schermo per visualizzare un cerchio in ogni punto toccato. Avete a disposizione la classe `CircleTouch` che permette di creare oggetti grafici da visualizzare sullo schermo. La classe ha i seguenti metodi

- costruttore `CircleTouch(Context c, int id)`: crea un nuovo `CircleTouch`, con identificatore `id`
- metodo `sposta(int x, int y)`: aggiorna le coordinate dell'oggetto
- metodo `onDraw(Canvas c)`: disegna il cerchio.

Assumendo che l'oggetto `main_window` sia il layout esterno al quale aggiungere e rimuovere gli oggetti `CircleTouch`, indicare le principali istruzioni da inserire nel seguente frammento di codice:

<pre>switch (event.getActionMasked()) { case MotionEvent.ACTION_DOWN: case MotionEvent.ACTION_POINTER_DOWN: CircleTouch circle = new CircleTouch(getApplicationContext(), event.getPointerId(event.getActionIndex())); main_window.addView(circle); circle.invalidate(); break; case MotionEvent.ACTION_UP: case MotionEvent.ACTION_POINTER_UP: CircleTouch circle = new CircleTouch(getApplicationContext(), event.getPointerId(event.getActionIndex())); main_window.removeView(circle.getId()); main_window.invalidate(); break; }</pre>	<pre>case MotionEvent.ACTION_MOVE: CircleTouch circle = new CircleTouch(getApplicationContext(), event.getPointerId(event.getActionIndex())); circle.sposta((int) event.getX(), (int) event.getY()); circle.invalidate(); break;</pre>
---	--

SENSORI

Si descriva l'utilizzo di un sensore spiegando cosa si deve fare per utilizzarlo. Arricchire la spiegazione con frammenti di codice.

I sensori consentono di misurare il movimento, l'orientamento e varie condizioni ambientali, sono in grado di fornire dati grezzi con elevata precisione e accuratezza e sono utili se si desidera monitorare il movimento o il posizionamento tridimensionale del dispositivo o se si desidera monitorare i cambiamenti nell'ambiente circostante ad un dispositivo. Android supporta tre grandi categorie di sensori:

- **Sensori di movimento**, misurano le forze di accelerazione e le forze di rotazione lungo tre assi, categoria che comprende accelerometri, sensori di gravità, giroscopi e sensori vettoriali rotazionali.
- **Sensori ambientali**, misurano vari parametri ambientali, come temperatura e pressione dell'aria ambiente, illuminazione e umidità, categoria che comprende barometri, fotometri e termometri.
- **Sensori di posizione**, misurano la posizione fisica di un dispositivo, categoria che comprende sensori di orientamento.

Per utilizzare un sensore in un'activity, occorre seguire gli step sottostanti:

1. L'activity deve implementare **SensorEventListener**:

```
public class SensorActivity extends Activity implements SensorEventListener { ... }
```

2. In **onCreate()** istanziamo il **sensorManager**, istanziamo il sensore e controlliamo se esso esiste (es. il sensore **TYPE_LIGHT** per la luminosità dell'ambiente):

```
private SensorManager sensorManager;
private Sensor sensorLight;
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
sensorLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
if (sensorLight != null) {
    // success
} else {
    // failure
}
```

3. Importante è, in **onResume()** e **onPause()**, rispettivamente la registrazione e la deregistrazione del sensor listener:

```
@Override
protected void onResume() {
super.onResume()
sensorManager.registerListener(this, lightSensor, SensorManagerSENSOR_DELAY_NORMAL);
}
@Override
protected void onPause() {
super.onPause();
sensorManager.unregisterListener(this);
}
```

4. Eseguire l'override di metodi **onAccuracyChanged()** e **onSensorChanged()**, dei listener invocati rispettivamente quando cambia la precisione del sensore e quando viene effettuato un campionamento:

```
@Override
public final void onAccuracyChanged(Sensor sensor, int accuracy) {
// do something when accuracy changes
}
@Override
public final void onSensorChanged(SensorEvent event) {
// values are stored in event.values array
}
```

Quando si registra il listener di un sensore è possibile selezionare la velocità di campionamento da utilizzare:

- SENSOR_DELAY_NORMAL (0,2sec)
- SENSOR_DELAY_GAME (0,02sec)
- SENSOR_DELAY_UI (0,06sec)
- SENSOR_FASTEST (0sec)

Si discuta dei vantaggi e svantaggi di queste varie possibilità e di quale accortezza deve avere il programmatore per un app che utilizza i sensori.

La possibilità di selezionare quattro opzioni per la velocità di campionamento dei sensori si basa su due fattori: performance dell'applicazione e durata della batteria del dispositivo. Per "performance" s'intende lo scopo dell'applicazione, ed è opportuno eseguire un trade-off tra i due fattori specificati. Un vantaggio che riguarda la performance indica che un'alta frequenza di campionamento implica una maggiore precisione dei dati ricavati; tuttavia, i sensori richiedono un uso elevato della batteria. Ad esempio, con SENSOR_DELAY_NORMAL abbiamo una bassa precisione ed un utilizzo ridotto della batteria rispetto a SENSOR_DELAY_GAME, che ha un'alta precisione ed un utilizzo elevato della batteria.

L'accortezza che il programmatore deve avere quando sviluppa un'app che utilizza i sensori è, di conseguenza, la gestione ottimizzata dei sensori: questi vanno registrati e deregistrati programmaticamente.

NOTIFICHE

Quali sono i principali modi per fornire notifiche all'utente? Si fornisca una breve descrizione.

- **Toast:** Un Toast è una piccola notifica popup che viene utilizzata per visualizzare informazioni sull'operazione che abbiamo eseguito nella nostra app. Toast mostrerà il messaggio per un breve periodo di tempo e scomparirà automaticamente dopo un timeout. Generalmente, la notifica Toast in Android verrà visualizzata con un testo semplice.
- **Dialog:** una piccola finestra che richiede all'utente di prendere una decisione o inserire informazioni aggiuntive. Una finestra di dialogo non riempie lo schermo e viene normalmente utilizzata per eventi modali che richiedono agli utenti di eseguire un'azione prima di poter procedere.
- **Notification Area (Status Bar):** è un messaggio che Android visualizza all'esterno dell'interfaccia utente dell'app per fornire all'utente promemoria, comunicazioni di altre persone o altre informazioni tempestive dall'app. Gli utenti possono toccare la notifica per aprire l'app o eseguire un'azione direttamente dalla notifica. Le notifiche possono apparire brevemente in una finestra mobile denominata notifica heads-up. Questo comportamento è in genere per le notifiche importanti che l'utente dovrebbe conoscere immediatamente e appare solo se il dispositivo è sbloccato.

Le notifiche in un app android sono brevi messaggi per l'utente visualizzati in vari modi. Si elenchino le tipologie di notifiche conosciute, possibilmente con dei frammenti di codice che mostrano il loro utilizzo.

- **Toast:** `Toast.makeText(getApplicationContext(), "Toast!", Toast.LENGTH_LONG).show();`
- **Dialog:**

```
DialogInterface.OnClickListener dialogClickListener = new DialogInterface.OnClickListener() {  
    public void onClick(DialogInterface dialog, int which) {  
        switch (which) {  
            case DialogInterface.BUTTON_POSITIVE:  
                // to do  
            case DialogInterface.BUTTON_NEGATIVE:  
                // to do  
        }  
    }  
};  
  
AlertDialog.Builder builder = new AlertDialog.Builder(this);  
builder.setMessage("Stai per ripartire da capo. Sei sicuro?")  
.setPositiveButton("Si", dialogClickListener)  
.setNegativeButton("No", dialogClickListener).show();
```

- **Notifica:**

```
Notification.Builder notificationBuilder = new Notification.Builder( getApplicationContext()).setTicker("Esempio")  
NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);  
mNotificationManager.notify(1,notificationBuilder.build());
```

Che cosa è un Toast customizzato? Si spieghi come implementare un Toast customizzato.

Un toast è una piccola finestra che occupa una porzione dell'UI usata per fornire informazioni all'utente durante l'interazione con l'applicazione; spesso viene utilizzato per informare l'utente circa l'esito o il progresso di una computazione. In Android, un toast di default è bianco, semi-trasparente ed ha dimensioni e layout del testo ben definiti. Mentre viene mostrato un toast, l'activity corrente resta visibile ed interattiva: essi spariscono automaticamente dopo alcuni secondi.

Android dà la possibilità di creare un toast customizzato. Per fare ciò, dapprima bisogna fornire il layout del toast in un file XML. In seguito, programmaticamente:

- creiamo un'istanza di Toast;
- impostiamo vari attributi del toast (gravity, duration, ...) e ne impostiamo il layout con il metodo `setView()`, a cui passiamo l'`inflate` di tale layout;
- lo mostriamo con il metodo `show()`.

```
public void showCustomToast(View v) {  
    Toast toast = new Toast(getApplicationContext());  
    toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);  
    toast.setDuration(Toast.LENGTH_LONG);  
    toast.setView(LayoutInflater.inflate(R.layout.custom_toast,null));  
    toast.show();  
}
```

PERMESSI

Si descriva il meccanismo dei permessi spiegando la differenza fra permessi normali e permessi pericolosi. Si metta in evidenza la gestione dei permessi in gruppi spiegando come vengono gestiti tali gruppi.

Lo scopo di un permesso è proteggere la privacy di un utente Android. Le app Android devono richiedere l'autorizzazione per accedere ai dati sensibili dell'utente (come contatti), nonché a determinate funzionalità del sistema (come videocamera e Internet). A seconda della funzione, il sistema potrebbe concedere l'autorizzazione automaticamente (*Permessi normali*) o richiedere all'utente di approvare la richiesta (*Permessi pericolosi*).

- I **permessi normali** riguardano aree in cui l'app deve accedere a dati o risorse al di fuori della sandbox dell'app, ma in cui esiste un rischio minimo per la privacy dell'utente o il funzionamento di altre app. Ad esempio, l'autorizzazione per impostare il fuso orario è un'autorizzazione normale. Se un'app dichiara nel manifest che ha bisogno di una normale autorizzazione, il sistema concede automaticamente tale autorizzazione al momento dell'installazione. Il sistema non richiede all'utente di concedere le autorizzazioni normali e gli utenti non possono revocare tali autorizzazioni (ad esempio BLUETOOTH, FLASHLIGHT, INTERNET, NFC).
- I **permessi pericolosi** coprono aree in cui l'app richiede dati o risorse che coinvolgono le informazioni private dell'utente o che potrebbero potenzialmente influire sui dati memorizzati dell'utente o sul funzionamento di altre app. Ad esempio, la capacità di leggere i contatti dell'utente è un'autorizzazione pericolosa. Se un'app dichiara di aver bisogno di un'autorizzazione pericolosa, l'utente deve concedere esplicitamente l'autorizzazione all'app. Fino a quando l'utente non approva l'autorizzazione, l'app non può fornire funzionalità che dipendono da tale autorizzazione. I permessi pericolosi per le API < 23 si approvano quando si installa l'app, mentre da 23 in poi a runtime. Quando l'app richiede un permesso pericoloso, se ha già un permesso per lo stesso gruppo allora viene concesso automaticamente, altrimenti viene richiesto all'utente (dialog box) il permesso per il gruppo (CALENDAR, CAMERA, CONTACTS, LOCATION)

Un'app utilizza 3 activity, quella principale, MainActivity, che viene lanciata alla partenza dell'app e due activity secondarie, ActivityA e ActivityB. L'activity A ha necessità di accedere al GPS mentre l'activity B ha necessità di scattare fotografie. Mostrare le parti rilevanti del file Manifest.xml.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.CAMERA"/>
```

CONTENT PROVIDER, BROADCAST E SERVICES

Le activity sono una delle 4 componenti principale di un'app. Quali sono le altre 3? Si fornisca una breve descrizione.

Le 4 componenti fondamentali di Android sono:

- **Activity:** //
- **Broadcasts:** possono essere utilizzate come sistema di messaggistica tra le app e al di fuori del normale flusso di utenti. Tuttavia, è necessario fare attenzione a non abusare dell'opportunità di rispondere alle trasmissioni ed eseguire lavori in background che possono contribuire a rallentare le prestazioni del sistema. Ad esempio, il sistema Android invia trasmissioni quando si verificano vari eventi di sistema, ad esempio quando il sistema si avvia o il dispositivo inizia a caricarsi.
- **Content Providers:** fornisce i dati da un'applicazione ad altre su richiesta. Tali richieste sono gestite dai metodi della classe ContentResolver. Un Content Providers può utilizzare diversi modi per archiviare i propri dati ad esempio in un database, in file o persino su una rete.
- **Services:** è un componente dell'applicazione che può eseguire operazioni di lunga durata in background e non fornisce un'interfaccia utente. Un altro componente dell'applicazione può avviare un servizio e continua a essere eseguito in background anche se l'utente passa a un'altra applicazione. Ad esempio, un servizio può gestire transazioni di rete, riprodurre musica, eseguire operazioni di I / O su file o interagire con un fornitore di contenuti, tutto da sfondo.