Appunti Mobile Programming

[Disclaimer: originariamente gli appunti includevano soltanto argomenti trattati sulle slide, ma sono state integrate diverse nozioni teoriche utili al fine di rispondere alle domande dello scritto]

Sommario

00 – Link utili

- Google
  - http://developer.android.com
  - http://developer.android.com/guide
  - http://developer.android.com/training
- Books: BigNerd Ranch (in inglese) Ultima edizione (5a) usa Kotlin al posto di Java
  - http://www.bignerdranch.com/
- Cursera, ottimo video corso (in inglese)
  - https://www.coursera.org/course/android

01 – Introduzione

L'architettura Android può essere suddivisa in diversi layer, dall'alto in basso: Applications, Java API, Librerie e Android Runtime, Hardware Abstraction Layer e Kernel Linux.

- Il **kernel linux** fornisce i servizi di base del sistema operativo, come filesyste e gestione di memoria, processi ed interfaccia di rete, ed i servizi specifici per android, come gestione della batteria e della memoria condivisa e low memory killer.
- L'**hardware abstraction layer,** o **HAL,** è un insieme di interfacce standard per esporre le capacità hardware al livello superiore, come audio, bluetooth e fotocamera.
- Le app possono essere scritte in Java e/o Kotlin, vengono compilate in file *JavaBytecode* e, tramite un tool chiamato DX, vengono poi trasformate in un singolo file *Dex Bytecode.* Il file classes.dex (Dalvik Machine EXecutable) ottenuto contiene anche tutti i file di dati necessari e viene eseguito dalla *ART Virtual Machine.* Android Runtime, o ART, è una VM specifica per sistemi Android e la si usa a partire da 5.0 API level 21, mentre Dalvik per quelle precedenti alla 21. Sono presenti molte librerie native in quanto usate da svariate componenti, troviamo ad esempio webkit, libc, openGL ES, ...
- Il framework **Java API** racchiude funzionalità del SO Android in molteplici API: *view system* fornisce gli elementi di base per le interfacce utente, come le icone; i *content providers* consentono di accedere a dati di altre app, come i contatti in rubrica; il *package manager* gestisce l'installazione delle app sul dispositivo mobile; l'*activity manager* gestisce il ciclo di vita delle applicazioni e consente di passare da una all'altra; esistono altri managers, come location, notification, resource, telephony e window.
- Il layer delle **system apps** contiene, inizialmente, le applicazioni già presenti nel sistema, come la home, i contatti, il telefono, ecc...

L’organizzazione dei file in un progetto Android è caratterizzata da una struttura ad albero in cui la prima cartella (la radice) è denominata "app" e presenta al suo interno le seguenti sub-directories, il cui ordine è importante:

- **manifests,** che contiene il file AndroidManifest.xml che racchiude le informazioni essenziali dell’app. Nello specifico, ci consente di definire Packages, API, dettagli sulle autorizzazioni, ecc... Infine, troviamo anche le informazioni su nome, icona e versione dell'applicazione;
- **java,** che contiene il codice sorgente dell’applicazione, organizzato in vari packages ed utilizzato per costruire l’app;
- **res,** che contiene le risorse dell’app, ci sono al suo interno altre directory, come "layout" che fornisce la struttura statica dell’app (in xml), "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.

01B – Android Developer Tools

[breve elenco di roba da installare + come mettere la modalità developer sul telefono]

Gli oggetti della classe View hanno dei metodi listeners che si attivano quando si verifica un evento specifico, come un pulsante che viene premuto.

[esempio di hello world]

02 – Layouts

I **layout** definiscono l'aspetto grafico dell'interfaccia utente e possono essere definiti con file XML o in modo programmatico. Il primo è facile da specificare e porta un maggiore disaccoppiamento tra UI e logica di applicazione, ma impone staticità; il secondo è facile da adattare, ma impone una gestione del layout all'interno del codice, che porta a meno leggibilità.

In generale, i due metodi non sono mutualmente esclusivi, ma ci sono alcune situazioni in cui è possibile usarne solo uno, ad esempio per layout completamente statici che possono essere scritti usando soltanto un file XML. È invece necessario usare layout creati soltanto programmaticamente in contesti dinamici in cui bisogna creare interfacce sulla base di un numero variabile di elementi o quando si vuole creare componenti personalizzate sulla base di qualche scelta dell'utente.

Un **ViewGroup** è un raggruppamento di elementi, sia di base che di altri gruppi. Ne troviamo diverse tipologie, come Linear (orizzontale o verticale), Relative, Grid e Frame.

Ogni elemento supporta degli attributi utili a specificarne l'aspetto e la posizione e a fornirne informazioni. È possibile aggiungere nuovi identificatori tramite '@+', dove @ indica che il resto della stringa deve essere interpretato e + specifica che stiamo creando un identificatore; la sua assenza significherebbe che stiamo facendo riferimento a qualcosa di esistente.

Ogni view ha dei parametri di layout appropriati per il ViewGroup a cui appartiene: alcuni sono comuni, altri specifici. Tra i più comuni applicabili a più layout, oltre all'id, abbiamo:

- "**android:layout_height**"/"**android:layout_width**": specificano altezza/larghezza di base della View, attributi obbligatori per qualsiasi View all’interno del layout.
- "**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; può essere specificato per ogni lato (Top, Bottom, Left, Right).
- "**android:padding**": utilizzato per generare spazio tra il contenuto di un widget ed il suo bordo; anche qui si può specificare per ciascun lato.
- "**android:gravity**": specifica come un oggetto deve posizionare il suo contenuto dentro un componente.

Tra quelli specifici di alcuni layout troviamo invece:

- "**android:orientation**": usato per specificare, in un LinearLayout, come vadano disposti gli elementi ('vertical' o 'horizontal').
- "**android:layout_weight**": usato per specificare, in un LinearLayout, come distribuire il peso, cioè lo spazio disponibile, tra i widget figli.
- "**android:layout_alignParentTop**" e "**android:layout_alignParentBottom**": usati per specificare, in un Relative Layout, dove posizionare il widget rispetto ai bordi del genitore.
- "**android:layout_below**" e "**android:layout_above**": usati per specificare, in un Relative Layout, per posizionare un widget sopra o sotto un altro.

Una **View** è un rettangolo la cui posizione si calcola dall'angolo in alto a sx ed è relativa al parent e di cui si specificano larghezza ed altezza.

Indichiamo con **screen size** la grandezza reale del display, ad esempio 4", mentre con **screen density** quanti pixel ci sono nell'unità di area: sono raggruppati in low, medium, high ed extra high; ad esempio 240dpi = 240 dot-per-inch. Con px si indicano i pixel reali, ad esempio 240dpi * 4" = 960 pixel, mentre i dip, dp o density indipendent pixels, calcolano la dimensione su una densità di 160dpi. La misurazione tramite dpi è necessaria per preservare la dimensione visibile dell'interfaccia utente su schermi con densità diverse (ad esempio, impostando 100px anziché 100dpi avrei due rappresentazioni molto diverse se uno schermo avesse una certa densità X ed un altro avesse invece 2X).

Le unità di misura a nostra disposizione sono:

- *dp*, density-independent pixels, si tratta di un'unità astratta basata sulla densità fisica dello schermo;
- *sp*, scale-independent pixels, scalato in base alle preferenze dell’utente sulla grandezza del font;
- *pt*, points (1/72 di inch), corrispondono ad 1/72 di pollice in base alle dimensioni fisiche dello schermo;
- *px*, real pixels, unità di base del display che variano in dimensione in base alla densità dello schermo;
- *mm*, millimetri;

- *in*, inches.

Una buona app dovrebbe fornire alternative per i Drawable, cioè gli oggetti da disegnare: è buona norma prevedere 4 versioni per ciascuna immagine, ovvero: 36*36 per display con densità low; 48*48 medium, 72*72 high, 96*96 extra high.

Un **LinearLayout** posiziona gli elementi uno dopo l'altro verticalmente oppure orizzontalmente ed occupa tutto lo spazio disponibile: se si desidera lasciare spazi vuoti bisogna creare elementi fittizi.

In un **RelativeLayout** la posizione degli elementi è relativa al layout padre ed agli altri elementi del layout. Una sua versione alternativa è il ***ConstraintLayout,*** che viene usata dall'editor grafico per specificare la posizione di nuovi oggetti tramite vincoli rispetto a quelli esistenti.

Un **GridLayout** visualizza un insieme di elementi il cui numero è variabile: nel caso non rientrino nello schermo, viene mostrata una sola parte e c'è uno scroll. Lo stesso discorso si applica ad una **List View**.

Alcuni dei widget principali sono TextView, Button, TextEdit, ImageView, CheckView e RadioButton.

Indichiamo con il termine "View Contenitore" quelle View che possono contenere al loro interno altri elementi (cioè i Layout) e con "View Terminale" quelle View che invece non possono contenere altri elementi al loro interno (i widget). Ne deriva che la radice dell'albero delle View è obbligatoriamente un contenitore e che tutte le foglie sono delle view terminali.

02B – Android Studio Debugger

Da slide 80 a 92, spiega come usare il debugger.

03 – List View

ListView è un widget specifico per le liste che divide l'area disponibile in varie sezioni: gli elementi sono memorizzati in un array, che solitamente è più grande dello spazio disponibile nel widget, ed è possibile scorrere per visualizzare gli altri elementi. Un adapter è un oggetto che gestisce e fornisce gli elementi da visualizzare in base allo scorrimento effettuato dall'utente. Lo si assegna ad un ListView tramite il metodo setAdapter.

È possibile avere liste semplici, in cui ogni elemento è una stringa, o liste personalizzate, in cui gli elementi hanno un proprio layout con dei sottoelementi. In quest'ultimo caso, il click è su tutto l'elemento, ma è possibile definere liste con click multiplo per cliccare sui singoli sottoelementi.

Per personalizzare gli elementi di un ListView dobbiamo usare un Adapter customizzato, cioè un Adapter che consente di adattare 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 invocato ogniqualvolta 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 gestire il click multiplo è inoltre necessario implementare manualmente dei listeners ad-hoc: la sua posizione nell'array può essere ricavata tramite getTag dopo averla settata in precedenza con setTag.

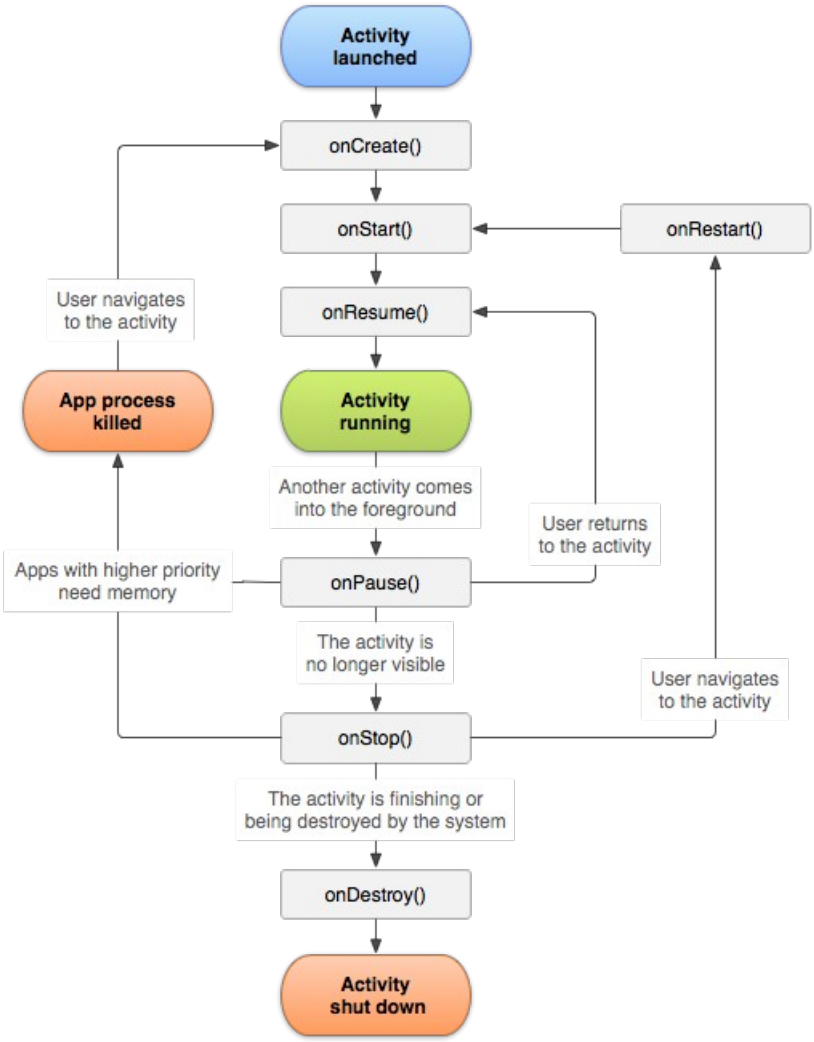
04 – Ciclo di vita

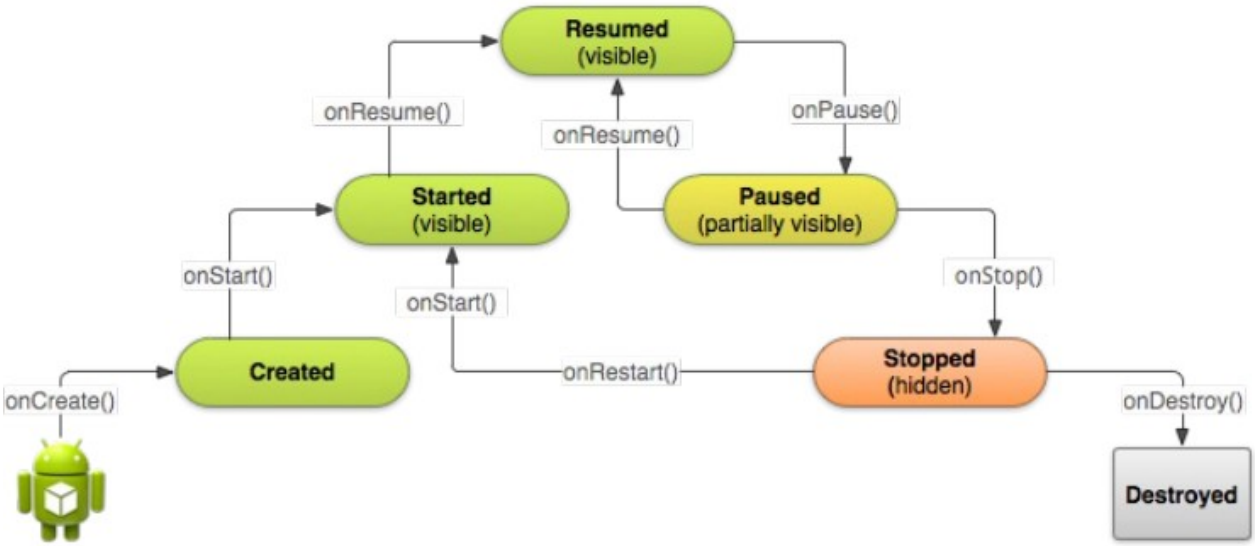
Ogni Activity è caratterizzata da un proprio ciclo di vita, scandito dalle seguenti azioni:

- **onCreate():** Inizializzazione; prevede operazioni che vanno eseguite una sola volta nel ciclo di vita dell'applicazione, come associare l'activity ad un ViewModel.
- **onStart():** Attività visibile ma non interattiva; prevede attività che preparano l'attività ad andare in foreground, come l'inizializzazione dell'interfaccia utente.
- **onResume():** Attività interattiva; prevede solitamente l'inizializzazione delle componenti necessarie all'applicazione.
- **onPause():** Attività parzialmente visibile; prevede il rilascio delle componenti inizializzate con onResume().
- **onStop():** Attività nascosta; prevede il rilascio di tutte le risorse non necessarie quando l'app non è più in foreground e può essere usata per delle operazioni dispendiose di shutdown, come il salvataggio dei dati.
- **onDestroy():** Risorse liberate.

Quando l'utente preme il pulsante 'home' vengono chiamate onPause e onStop, mentre quando si torna all'attività vengono chiamate onRestart, onStart e onResume. Quando l'utente ruota il dispositivo, l'attività viene prima eliminata e poi ricreata: per non perderne lo stato bisogna usare il metodo onSaveInstanceState, in modo da recuperarlo successivamente tramite onCreate.

La rotazione del dispositivo non influisce necessariamente su un'activity in background, in quanto il sistema potrebbe terminarla soltanto se in mancanza di risorse: potrebbe invece essere terminata e ricreata nel caso l'utente vi faccia ritorno.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down


Resumed (visible)
onResume()
onResume()
onPause()
Started (visible)
Paused (partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped (hidden)
onCreate()
onDestroy()
Destroyed

La presenza di più metodi per scandire le molteplici fasi del ciclo di vita di un'activity consente una gestione più fine delle risorse impiegate ed una loro liberazione 'a specchio' una volta che l'app smette di essere in foreground. Il metodo onCreate() si occupa di quelle operazioni da eseguire una sola volta nel ciclo di vita, onStart() di quelle che preparano l'attività ad andare in foreground e diventare interattiva ed infine onResume() di quelle inerenti alle altre componenti da usare. Con onPause() liberiamo proprio quest'ultime, in quanto l'app, ora parzialmente visibile, potrebbe non necessitare più di componenti quali, ad esempio, la fotocamera. Successivamente, onStop() libera le risorse non necessarie quando l'app non è più in foreground ed infine, se l'utente non riapre l'app, viene chiamato onDestroy() per rimuovere le ultime risorse. Compattare tutto in due soli metodi, onEsecuzione() e onFine(), non potrebbe garantire nulla di tutto ciò e priverebbe inoltre l'utente della possibilità di ritornare sull'app tramite onRestart() dopo che è stato chiamato onStop(). Un'altro svantaggio sarebbe quello di incorrere in arresti anomali quando l'utente ruota lo schermo o quando, ad esempio, riceve una telefonata e passa ad un app diversa.

Il periodo di visibilità va dalla chiamata onStart() alla chiamata onStop(), mentre il periodo di foreground va da onResume() a onPause().

Le seguenti chiamate vengono effettuate in modo automatico: onStart(), onResume(), onStop(), onDestroy(), onRestart(); richiedono dell'intervento dell'utente o del sistema: onCreate() e onPause().

Sono indicati come stati transienti (in cui l'activity resta pochissimo tempo) Created, Started e Paused, mentre sono indicati come duraturi (l'activity vi può rimanere per molto tempo) Resumed e Stopped.

05 – Cambi di configurazione

La configurazione del device coinvolge un gran numero di parametri:

- Screen orientation (portrait, landscape).
- Layout direction: da sinistra a destra, da destra a sinistra.
- Available width, height.
- Screen size (small, normal, large, xlarge).
- Round screen (e.g. orologio).
- UI mode (car, desk, television, appliance,watch, vrheadset).
- keyboard availability (keysexposed, keyshidden, keyssoft).
- altre: configuration qualifier names.

Quando avviene un cambio di configurazione, il sistema operativo distrugge e ricrea l'attività in esecuzione, permettendo all'app di adattarsi al meglio alla nuova configurazione. Tramite onConfigurationChanged() è inoltre possibile gestire in proprio il cambiamento.

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.putStringArrayList(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), …

Un'alternativa recente a tale operazione è usare un nuovo 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.

06 – Backstack

Un'app è normalmente fatta di più activity, ciascuna con un compito specifico, ed è possibile che un'attività ne lanci un'altra, anche di un app diversa. Quando più attività possono coesistere, vengono organizzare in un **backstack,** cioè una pila (struttura LIFO), che in genere parte dall'Home screen: quando l'utente clicca un'icona, l'applicazione viene portata in ***foreground;*** se vengono lanciate nuove attività quella corrente viene messa nel backstack e vi si può tornare premendo il pulsante Back. È utile notare che le attività non vengono mai riorganizzate e che è possibile fare in modo che un'activity non vi finisca memorizzata usando setFlag(FLAG_ACTIVITY_NO_HISTORY) al momento del lancio.

07 – Intent

Un **intent** è una descrizione astratta di un'operazione da svolgere e può essere eusato per diversi scopi: lanciare una nuova attività (startActivity), spedire l'intent in broadcast (broadcastIntent), comunicare con un servizio in background (startService o bindService) e trasferire dati tra attività. Le componenti principali sono:

- **Action:** l'azione da svolgere (ACTION_VIEW, ACTION_EDIT, ...).
- **Data:** i dati su cui operare, espressi come URI.

Altre parti comuni di un intent sono:

- **Category:** informazioni aggiuntive sull'azione da eseguire.
- **Type:** specifica esplicitamente il MIME type dei dati, che viene di norma dedotto.

- **Component:** specifica esplicitamente l'attività da eseguire, altrimenti dedotta dalle altre informazioni. Senza abbiamo una risoluzione implicita, altrimenti è esplicita.
- **Extras:** informazioni addizionali in coppie chiave-valore.
- **Flags:** indicazioni su come gestire l'intent (es. evitare di aggiungerlo al backstack con FLAG_ACTIVITY_NO_HISTORY).

Per condividere dati tra più activities possiamo dichiarare statiche le risorse che vogliamo condividere, per poi accedervi tramite getters e setters, ma un modo più comune è quello di ricorrere agli Intent, che offrono più flessibilità. Possiamo creare un nuovo Intent a cui forniamo come primo parametro il contesto dell'attività chiamante e come secondo l'activity che vogliamo chiamare e che riceverà i dati. Dopo aver inserito i valori tramite putExtra(name, value), chiamiamo start(activity). Nella seconda activity invece usiamo getIntent() per creare l'intent e poi getExtra(name) per ottenere il valore inserito. Il metodo spiegato è detto '**Direct Intent**', in quanto viene usato l'Intent come contenitore dei dati, ma è possibile seguire una seconda strada, quella dei Bundle: l'Intent conterrà al suo interno soltanto un oggetto Bundle, che si occuperà di mantenere tutti i dati al suo interno. L'uso di un Bundle rende il codice leggermente più pesante, ma lo rende anche più facile da leggere e gestire; inoltre, i Bundle consentono una serializzazione più pulita dei dati prima del loro invio, se necessario.

08 – Permessi

I **permessi** sono meccanismi di protezione per le risorse e i dati, a cui si può accedere solo tramite un consenso esplicito dell'utente. Limitano l'accesso ad informazioni sensibili, servizi con costi e risorse di sistema e bisogna dichiarare nel manifesto quali permessi si vuole chiedere. Li dividiamo in:

- **normali,** se concessi automaticamente senza chiedere nulla all'utente;
- **pericolosi,** se vanno approvati al momento dell'installazione (API<23) o a runtime (API>=23).

È sempre possibile cambiare la propria decisione tremite le autorizzazioni nel menù delle app installate, motivo per cui il programmatore deve sempre controllare che il permesso ci sia tramite ContextCompat.checkSelfPermission(...).

I permessi sono rappresentati come delle stringhe ed appartengono a dei macro-gruppi, ad esempio Calendar contiene Read_Calendar e Write_Calendar: quando un app chiede un permesso pericoloso, questo viene concesso automaticamente se essa ha già un permesso per lo stesso gruppo oppure viene richiesto all'utente il permesso per l'intero gruppo.

09 – Threads

I **threads** consentono la computazione parallela all'interno di un processo: ciascuno ha il proprio program counter ed il proprio stack e condivide con tutti gli altri l'heap e la memoria statica. Gli oggetti java.lang.Thread implementano l'interfaccia runnable, per cui devono avere il metodo run(); altri

metodi utili sono start(), sleep(long time), wait() e notify(). Per usare un thread bisogna creare l'oggetto e chiamare il metodo start() del thread, che a sua volta chiamerà run(). Una restrizione di Android impone che soltanto il main thread interagisca con l'interfaccia utente, pertanto questi si occupa, ad esempio, di gestire sia tutti gli input dell'utente che tutti gli output. Per facilitare l'interazione tra main e background thread si usano le *Async task,* che eseguono il task e notificano sullo stato di avanzamento. Abbiamo una classe generica AsyncTask<Params, Progress, Result>:

- ***Params*** indica il tipo di dati per il lavoro che deve svolgere il thread;
- ***Progress*** il tipo di dati usato per lo stato di avanzamento;
- ***Result*** il tipo di dati per il risultato del task.

Tra i metodi principali troviamo:

- ***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... 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 ed il risultato del calcolo deve essere restituito da questo passaggio. Si 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... values),*** 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 result),*** richiamato sul thread dell'interfaccia utente, ricevendo l'output di doInBackground come parametro.

10 – Fragments

Un **fragment** è una porzione dell'interfaccia utente ospitata da un'activity, che ne può mantenere vari inserendoli e rimuovendoli durante l'esecuzione. Possiamo pensarli come sub-activity con un ciclo di vita proprio, ma comunque legato a quello dell'activity. La porzione di UI occupata deve essere specificata nel layout e può essere definita dinamicamente. I fragments sono utili per creare interfacce dinamiche e adattabili a schermi di diverse dimensioni: se ad esempio un'app mostra un elenco di titoli di giornale e fornisce la possibilità di esaminarli singolarmente, uno schermo grande userebbe due frammenti entrambi visibili, mentre uno piccolo farà passare da uno all'altro. Dopo che l’activity viene stoppata e prima che essa viene distrutta, viene effettuato il detatch del frammento (metodo

onDetatch()). Una classe Fragment cura l’utilizzo dinamico del frammento nel corso dell’interazione con l’utente.

I metodi da implementare necessariamente per la sua creazione sono:

- ***onCreate():*** inizializza il frammento come in un activity;
- ***onCreateView():*** in cui si definisce il layout del frammento;
- ***onPause():*** che salva modifiche quando il frammento viene rimosso.

È possibile inserire un frammento sia staticamente tramite XML che dinamicamente tramite *FragmentManager* e *FragmentTransaction*. Poiché il backstack considera soltanto le activity, i frammenti vanno gestiti manualmente ed i cambiamenti effettuati vengono inseriti con il metodo *addToBackStack()*.

È buona norma non far comunicare due frammenti tra loro in maniera diretta, si dovrebbe invece preferire il passaggio per l'activity che li sta ospitando tramite delle interfacce di callback. I vantaggi nell'usare l'activity come intermediario sono:

- **Disaccoppiamento;** i frammenti non avrebbero una dipendenza diretta e potrebbero dunque restare modulari e facilmente riutilizzabili.
- **Manutenibilità;** l'activity 'mediatore' facilita modifiche future, riducendo l'impatto su ciascuna delle parti coinvolte.
- **Gestione del ciclo di vita,** in quanto i frammenti potrebbero non essere attivi allo stesso tempo.
- **Chiarezza del flusso di dati,** rendendo il codice più leggibile.

11 – Networking

La comunicazione via rete utilizza le ***socket*** di java.net a basso livello e le ***connessioni HTTP*** tramite la classe httpURLConnection a più alto livello (importanti i metodi openConnection e getInputStream); di quest'ultime si fa parsing tramite librerie come ***Jsoup.*** Gli indirizzi IP sono gestiti tramite la classe InetAddress, che manipola sia IPv4 che IPv6; la classe Socket crea il canale di comunicazione con il server. Il localhost è 10.0.2.2

La classe Jsoup permette di anche di estrarre singole parti dei documenti di cui viene passato il link, che da API 28 in poi deve essere https (oppure si abilita manualmente anche http): cercare di usare link http quando non abilitati porta al messaggio di errore 'cleartext not permitted'.

Per impostazione predefinita, da Android 3.0 (livello API 11) si 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.

12 – Data Storage

Esistono diverse opzioni di archiviazione dei dati:

- *Relativo all'app,* cioè in memoria interna o app-specific; viene rimossa alla disinstallazione.
- *Shared storage,* cioè file condivisibili con altre app, come media o documenti; richiede permessi specifici.
- *Preferences,* cioè coppie chiave-valore per dati semplici e privati a cui accedere tramite getter e setter.
- *Database SQLite* privati per l'archiviazione strutturata in tabelle. Per accedervi è fondamentale la classe ***Cursor,*** che permette di gestire dataset grandi senza caricarli in memoria accedendovi tramite metodi ottimizzati. È necessario chiuderlo (cursor.close()) per evitare memory leak.
- *Cache,* una directory per file temporanei che il sistema può eliminare in mancanza di spazio.
- Va inoltre notato che è possibile sia presente una *memoria esterna:* ha file pubblici (world-readable) e richiede il permesso di lettura e scrittura.

Oltre ai db, i due metodi principali per condividere informazioni tra le activity della stessa app sono:

- Le **SharedPreferences** sono delle preferenze (localizzate in opportuni file di minime dimensioni) condivise tra più activity, che consentono la memorizzazione, tipicamente, di valori primitivi attraverso setter e getter. Per ricavare le preferenze condivise, è possibile utilizzare il metodo getDefaultSharedPreferences(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. Bisogna fare attenzione a non confonderle con le preferenze semplici ottenibili tramite getPreferences(), che serve invece ad ottenere le preferenze della singola activity e dunque non condivise tra tutte le activity dell'applicazione.
- I **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 ricavarle 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.

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.

13 – Grafica

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:

- **Misurazione:** fase in cui viene fatta una visita top-down dell’albero per misurare width e height di ciascun elemento in modo ricorsivo. In questa fase si effettuano in realtà due misurazioni: una "temporanea" in cui ogni View, tramite LayoutParams, indica al padre quanto grande vorrebbe essere, ed una definitiva in cui il padre, tramite MeasureSpec, indica le dimensioni effettive dell'elemento.
    - In particolare, l'oggetto MeasureSpec codifica con un intero sia le dimensioni in pixel che una modalità, entrambe recuperabili con appositi getter. Con getMode si ottiene uno dei seguenti valori: ***exactly,*** nel caso di una dimensione precisa come match_parent o un valore fisso, ***at_most,*** se la view può essere al massimo di una certa dimensione come con wrap_content, ***unspecified,*** se non c'è restrizione e la view può decidere la sua dimensione come in uno ScrollView.
- **Posizionamento:** fase in cui viene fatta una visita top-down dell’albero per posizionare ciascun elemento dell'albero in modo ricorsivo.
- **Disegno:** fase in cui si disegna ogni view nel layout dell’interfaccia.

Poiché le visite dell'albero possono essere dispendiose in termini di calcolo e tempo, è possibile che durante l'esecuzione del metodo onCreate() chiamate come view.getWidth restituiscano 0 in quanto non è ancora possibile ricavare la misura della view.

Un'immagine può essere disegnata sia in un **oggetto View,** nel caso sia semplice e non necessiti di cambiamenti, o in un **oggetto Canvas,** se abbiamo bisogno di qualcosa di complesso che va riaggiornato. La classe Drawable rappresenta un oggetto che può essere disegnato, cioè immagini, colori (ColorDrawable), forme (ShapeDrawable), ecc... L'oggetto Drawable deve essere inserito nell'oggetto View, sia tramite XML che con View.setImageDrawable().

L’oggetto Canvas consente di disegnare una view customizzata in un determinato punto del layout. Lo step più impostante 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. Un esempio che richiede un oggetto Canvas può essere quello di un applicazione che mostra un pentagramma su cui è possibile posizionare e rimuovere delle note tramite click.

È possibile definire delle animazioni (rotazioni, traslazioni, trasparenze, ecc) da applicare alle immagini: sono descritte con file XML e le si applica alle ImageView tramite il metodo statico AnimationUtils.loadAnimation(Context context, int resource), che ritorna un oggetto Animation. Alla radice del file di layout deve essere posto un elemento <set> o, in alternativa, <objectAnimator> e <valueAnimator>: all'interno di <set> si possono porre ulteriori elementi <set> oppure singole animazioni tramite <animator> ed <objectAnimator>. Per usarle, c'è il metodo View.startAnimation(Animation animation).

14 – Multitouch

Un **MotionEvent** è un oggetto che rappresenta un movimento registrato da una periferica, come penna, mouse o dita sul display: il movimento è a sua volta rappresentato con ACTION_CODE, per il cambiamento avvenuto, ed ACTION_VALUES, per posizione e proprietà del movimento (come pressione, dimensioni ed orientamento dell'area di contatto). I multitouch display usano dei ***pointer*** per rilevare i singoli eventi: ciascuno ha un ID unico per tutto il tempo in cui esiste. Come ACTION_CODES troviamo:

- ACTION_DOWN, se un dito tocca lo schermo ed è il primo.
- ACTION_POINTER_DOWN, se un dito tocca lo schermo, ma non è il primo.
- ACTION_MOVE, se un dito che è sullo schermo si muove.
- ACTION_POINTER_UP, se un dito che è sullo schermo non lo tocca più.
- ACTION_UP, se l'ultimo dito sullo schermo viene alzato.

Metodi comuni sono:

- *getActionMasked(),* che restituisce l'Action Code dell'evento;
- *getPointerCount(),* che restituisce il numero di pointer coinvolti;
- *getActionIndex(),* che restituisce l'indice di un pointer;
- *getPointerID(int pointerIndex),* che restituisce l'ID del pointer specificato;
- *getX(int pointerIndex)* e *getY;*

- *findPointerIndex(int pointerId);*
- *View.onTouchEvent(MotionEvent e),* che notifica l'oggetto View restituendo un Boolean: true se è stato consumato, false altrimenti;
- *onTouch* viene invocato quando c'è un evento, prima che la View ne venga notificata; restituisce un Boolean: true se l'evento è stato consumato, false altrimenti (false implica che alla View 'non interessa' di questo evento).

La classe GestureDetector permette di riconoscere gesti fatti sul display, come pressione semplice o doppia e scorrimento, ed è possibile crearne di personalizzati.

15 – Media Player

MediaPlayer riproduce audio e video tramite un AudioManager, che controlla le sorgenti audio e l'output. Come sorgente dati possiamo avere risorse locali, in res/raw, da URI interni o da URL; questi ultimi devono ricorrere a *prepareAsynch* per essere riprodotti dopo *onPrepareListener.onPrepared().* Poiché c'è un solo canale di output, l'utilizzo da parte di più applicazioni può essere un problema. L'accesso contemporaneo viene gestito tramite l'audio focus, che deve essere richiesto per usare l'audio: quando l'app lo perde smette di suonare oppure abbassa il proprio volume.

16 – Sensori

Molti smartphones presentano diversi tipi di **sensori** che forniscono dati grezzi di accuratezza variabile in base alla qualità del dispositivo. Possiamo dividerli in tre grandi categorie:

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

L'oggetto SensorManager gestisce il tutto, comunicando quali sono i sensori disponibili e quali sono le loro caratteristiche e permettendoci sia di leggere i dati grezzi che usare dei Listeners sui cambiamenti. Come metodi principali troviamo:

- *onSensorChanged(),* per reagire ai cambiamenti del sensore;
- *onResume()* e *onPause()* per registrare e rilasciare il sensore per risparmiare batteria.

È possibile scegliere tra diverse velocità di campionamento quando si registra il listener di un sensore: un buon programmatore dovrebbe riuscire a trovare un compromesso tra reattività e consumo

energetico in base all'applicazione che sta sviluppando. Per decidere come impostare la velocità di campionamento, dovrebbe tenere in conto diversi fattori:

- bilanciare precisione e consumo, usando SENSOR_FASTEST solo se necessario e preferendo SENSOR_DELAY_GAME per giochi o SENSOR_DELAY_UI per UI reattive;
- gestire la concorrenza, elaborando i dati in un thread separato;
- ottimizzare la gestione dei dati, riducendo il numero di aggiornamenti elaborati e filtrando i dati ridondanti;
- disattivare i sensori quando non necessari, interrompendo il listener quando l'attività non è in foreground;
- gestire le variazioni tra dispositivi, testando su tipologie diverse.

17 – Notifiche

Le **notifiche** sono informazioni fornite all'utente al di fuori dell'interfaccia grafica dell'app e si dividono in:

- ***Toast:*** una piccola notifica popup che viene utilizzata per visualizzare informazioni sull'operazione che abbiamo eseguito nella nostra app. Un 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.
    - Per crearne uno sono necessari il messaggio da mostrare e la durata del popup e si usano i metodi makeText(context, text, duration) per creare il toast e show() per mostrarlo.
- ***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* (da API 21). Questo comportamento è in genere per le notifiche importanti che l'utente dovrebbe conoscere immediatamente e appare solo se il dispositivo è sbloccato.

Da API 26 abbiamo l'introduzione dei canali di notifica, controllabili dall'utente. Per la maggior compatibilità possibile conviene usare le classi NotificationCompact e NotificationManagerCompact.

18 – Alarms

Gli **alarms** permettono di eseguire operazioni basate sul tempo, anche se l'app è chiusa (quindi riattivandola) o il dispositivo è in sleep (quindi causando la ripresa delle attività o venendo gestito quando l'utente rimette la modalità normale). Ne troviamo due tipologie:

- ***Alarms imprecisi,*** di cui il sistema non garantisce l'esecuzione al tempo richiesto per minimizzare wakeups e l'uso della batteria; da API 31 vincolo a entro un'ora.
- ***Alarms precisi,*** che vengono eseguiti al tempo richiesto e da API 31 richiedono un permesso.

19 – Content Providers, Broadcast e Services

Oltre alle Activity, altre tre componenti fondamentali di Android, utili soltanto in alcune sitazioni, sono:

- **Broadcasts,** usati per inviare e ricevere messaggi dal sistema Android o da altre app. I messaggi possono essere globali, se ricevuti da tutti, o locali, se ricevuti solo all'interno dell'app, e richiedono un Sender, un Receiver ed un Intent. 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. Un esempio di broadcast è quello inviato dal sistema quando il dispositivo inizia a caricarsi, mentre un esempio di receiver è il cambio di connettività (come da wifi a dati mobili).
- **Content Providers,** usati da un'applicazione per accedere a dati di un'altra app oppure a dati strutturati presenti in un database. I client vi fanno riferimento tramite un URI ed è possibile avere molteplici providers sullo stesso DB in modo da fornire più viste. Esempi possono essere anche la galleria ed i contatti.
- **Services,** usati per task in background di lunga durata senza interfaccia utente, come un download. La classe Services usa i Services esistenti e consente di definirne di nuovi. Di default, il Service gira nel main thread dell'app che lo ha fatto partire e continua la sua esecuzione fino al termine dell'operazione richiesta; può capitare che termini volontariamente o che sia interrotto per mancanza di risorse. Un Service Unbound è indipendente e viene avviato con startService(), mentre uno Bound è collegato ad una componente che lo utilizza: fin quando ci sono client agganciati, il service sarà in esecuzione. Altri esempi oltre al download manager sono le app che consentono la riproduzione musicale.