Appunti Mobile Programming

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...

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. 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.

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.

Come unità di misura troviamo:

- *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;
- *mm*, millimetri;
- *in*, inches.

Un **LinearLayout** posiziona gli elementi uno dopo l'altro ed è possibile assegnare lo spazio ai figli tramite android:layout_weight. Se si desidera lasciare spazi vuoti bisogna inserire degli 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.

Altri widget legati ad i layout sono TextView, Button, TextEdit, ImageView, CheckView, RadioButton e molti altri.

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 gestisce e fornisce gli elementi da visualizzare in base allo scorrimento effettuato dall'utente.

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.

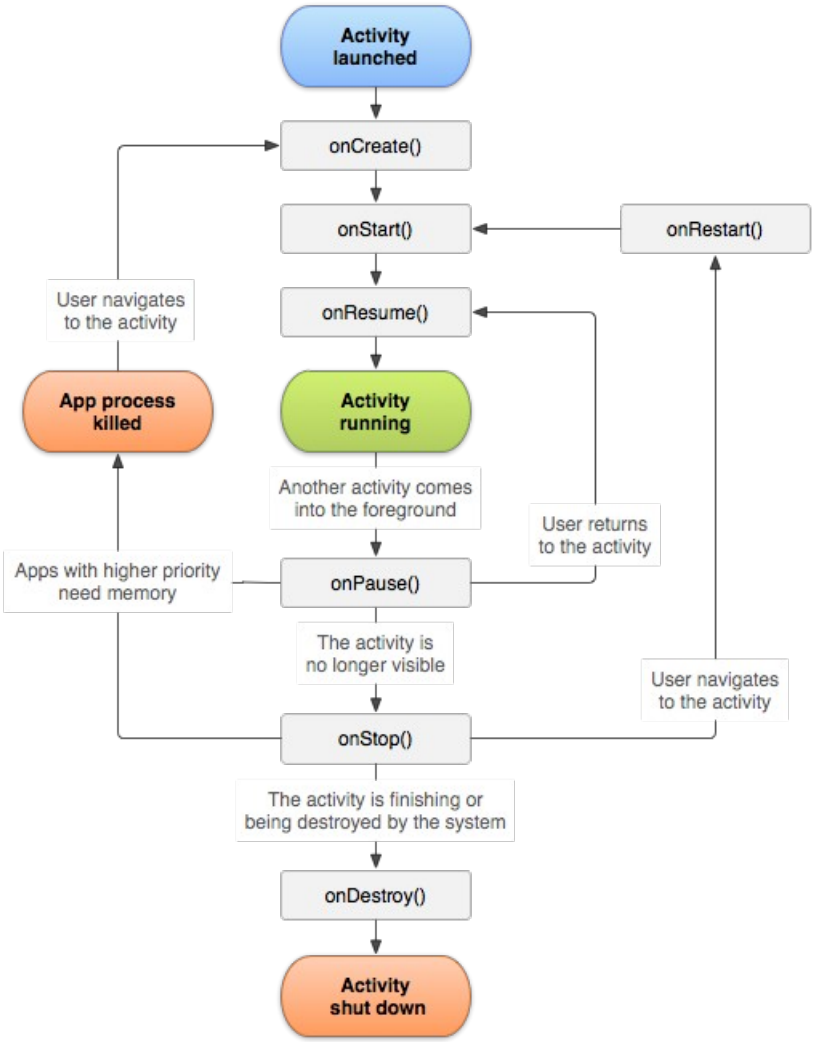
Per personalizzare gli elementi possiamo usare il file di layout e dei CustomAdapter. Per il click multiplo è invece necessario implementare manualmente dei listeners ad-hoc e per sapere quale sia la posizione dell'array che ci serve abbiamo setTag e getTag.

04 – Ciclo di vita

Ogni Activity ha un ciclo di vita, caratterizzato dalle seguenti azioni:

- **onCreate():** Inizializzazione; prevede operazioni che vanno eseguite una sola volta nel cilco 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 si usa onSaveInstanceState e lo si recupera poi con onCreate.

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.

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, 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.

07 – Intent

Un **intent** è una descrizione astratta di un'operazione da svolgere che permette di lanciare una nuova attività (startActivity), spedire l'intent in broadcast (broadcastIntent) e comunicare con un servizio in background (startService o bindService). 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 a cui vogliamo passare 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. 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(),* eseguito nel main thread prima di doInBackground();
- *doInBackground(Params... params),* eseguito su una lista variabile di parametri: restituisce un oggetto di tipo Result e può chiamare publishProgress(Progress... values);
- *onProgressUpdate(Progress... values),* eseguito nel main in risposta a publishProgress;
- *onPostExecute(Result result),* nel main, 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. 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*. I frammenti vanno gestiti manualmente ed i cambiamenti effettuati vengono inseriti con il metodo *addToBackStack().*

I frammenti possono comunicare con l'activity tramite delle interfacce di callback, migliorando il disaccoppiamento e la riusabilità.

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).

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.
- *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.

13 – Grafica

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().

È possibile definire delle animazioni da applicare alle immagini: sono descritte con file XML e le si applica alle ImageView tramite la classe Animation.

Seppur siano presenti molti widget, a volte è necessario crearne di personalizzati: il maggior controllo grafico però richiede di implementare metodi come *onMeasure, onLayout* e *onDraw.* Per ognuna occorre visitare l'albero delle view in modo top-down: i figli comunicano al padre, tramite LayoutParams, quanto vorrebbero essere grandi e dove vorrebbero essere posizionati, ed il padre, tramite MeasureSpec, comunica le dimensioni effettive.

14 – Multitouch

Un **MotionEvent** rappresenta un movimento registrato da una periferica, come penna, mouse o dira 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. 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);*
- *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** (ambiente, movimento, posizione) che forniscono dati grezzi la cui accuratezza dipende dalla qualità. SensorManager gestisce 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.

17 – Notifiche

Le **notifiche** sono informazioni fornite all'utente al di fuori dell'interfaccia grafica dell'app e si dividono in: *toast*, brevi messaggi temporanei, *dialog*, che richiedono interazione, e *notifiche*, status bar e cassetto delle notifiche. I messaggi possono apparire sulla barra di stato come icone, come cerchietto sull'icona e nel 'cassetto delle notifiche' con del testo aggiuntivo e sono visibili fin quando l'utente non li legge o l'app non li revoca. Da API 21 abbiamo le notifiche a comparsa (*heads-up*) per notifiche che sono ritenute urgenti, mentre 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.
- **Content Providers,** usati per accedere a dati strutturati presenti in un database. I client vi fanno riferimento tramite un URI ed è possibile averne molteplici sullo stesso DB.
- **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.