

STORIA DI ANDROID

I primi sistemi operativi per telefoni erano fortemente proprietari, ad esempio il sistema operativo di *blackberry* o *windows CE*. Si cercò di creare un sistema operativo per telefoni che non fosse fortemente proprietario. Nel 2007 si forma la *Open Handset Alliance* e 7 giorni dopo venne rilasciato il primo SDK (*Software Development Kit*) basato su linux 2.6. In realtà il software era già stato sviluppato da *Android Inc.* fondata da Andy Rubin, una start-up nata a Palo Alto (Silicon Valley) acquistata poi da Google nel 2005. Dal 2007 in poi vengono sviluppati numerosi S.O., nel 2015 viene rilasciato *Android 6 Marshmallow*, che portò dei cambiamenti significativi al sistema. I produttori hanno poco interesse nell'aggiornare i telefoni vecchi, facendone comprare di nuovi agli utenti, a causa di questo si è formata una frammentazione che rende difficile lo sviluppo e la retro-compatibilità. Ovviamente sono presenti delle API, identificate da un API Level. (livello minimo di cui necessitano per funzionare, target sulle quali sono state scritte e massimo oltre il quale non funzionano più). Google cerca di supportare anche le versioni vecchie di Android con librerie di compatibilità e i Google Play Services (funzioni incorporate in una libreria aggiornabile da Play Store).

AMBIENTI DI SVILUPPO

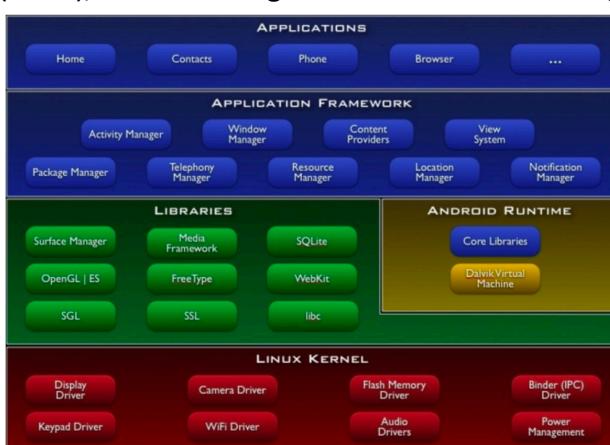
Sono presenti due livelli di sviluppo, programmazione nativa in C in ambiente Linux e standard Java in ambiente Android. Ci sono 3 ambienti di sviluppo per Android:

- 1) CLI (linea di comando - toolchain)
- 2) Eclipse con plug-in
- 3) Android Studio (versione customizzata di IntelliJ).

I componenti dell'ambiente di sviluppo tipici sono Il Java Development Kit (JDK), Android Development Tools (ADT), Android SoftwareDevelopmentKit (Android SDK) e altri. È possibile installare separatamente tutti i tools e collegarli tra di loro. Ma è molto più comodo usare un AndroidDevelopmentTool Bundle (ADT Bundle) su Eclipse. Android Studio è disponibile solo in versione bundle che include l'IDE e l'SDK. Android studio usa Gradle: un sistema di build dell'app avanzato. Un altro builder è Ant ma Gradle permette varianti multiple del prodotto finale, dipendenze remote (librerie accessibili tramite url) e compilazione dei campi del manifest. (problema: complessità nella configurazione). Il manifest è un file XML che contiene le informazioni principali riguardanti l'applicazione, i metadati, i vari permessi ecc.

Android studio ha integrato Lint, uno strumento per il checking di errori.

L'ADV manager è il punto di avvio per l'emulatore di Android. Esso è una macchina virtuale configurabile in due modi: su una immagine ARM l'app viene eseguita in modo interpretato (lento), su una immagine è x86 Atom viene eseguita l'app in modo nativo (veloce).



IL KERNEL

Alla base di Android è presente un kernel Linux, è un kernel completo con tutte le primitive, syscall UNIX e driver per i vari moduli. La maggior parte delle applicazioni eseguite sullo smartphone vengono caricate sulla macchina virtuale chiamata Dalvik (utilizzata dalla creazione di Android fino alla versione 4.4). E' una versione della JVM basata su registri e ottimizzata. Ma poiché Oracle (creatore di Java e della JVM) non ne ha il controllo, ha iniziato una causa legale, così Google ha deciso di creare la propria macchina virtuale a partire da Android 5, ha nome ART: pre-compila a tempo di installazione e non interpreta, producendo codice nativo e non più bytecode. ART crea un file ELF (Executable and Linkable Format).

Ogni app viene eseguita dal kernel linux in un processo separato che esegue Dalvik che esegue il bytecode. Ogni processo ha un user_ID distinto. Il controllo dei permessi è eseguito dalla VM. È possibile che applicazioni create dallo stesso user_ID condividano memoria e processo, ma altre applicazioni non possono accedere alla sua directory, il controllo di accesso alle risorse è fatto dal kernel. I veri utenti delle applicazioni (nel senso di UNIX) sono i programmatore delle app, il proprietario dello smartphone è quasi un device e vede l'app come un task, nel senso che non è proprietario di nulla e non fa nulla se non tramite app. Il risultato complessivo è un notevole grado di separazione e isolamento tra le applicazioni, ci possono però essere sempre degli exploit basati sull'ingegneria sociale come richiedere particolari permessi e usarli per scopi diversi da quelli pubblicizzati.

STRUTTURA DI UNA APPLICAZIONE ANDROID

3 principali fasi: sviluppo (creazione del sorgente), deployment (creazione apk e immissione su playstore) e esecuzione (da apk a processo in memoria).

La struttura di un progetto su Android Studio:

Manifest: (metadati sull'applicazione)

src - Java: Sorgenti

Assets: file arbitrari

Bin: risultato della compilazione

Res: risorse note a runtime tipo file xml e drawable

Libs: librerie native custom

Gradle Scripts: script per la configurazione della build.

Oltre alle applicazioni esistono due altre forme di progetto Android:

Librerie contenenti componenti destinati ad essere usati da altre applicazioni

Progetto Test che contiene codice usato per il testing di un'altra applicazione

Cosa contiene un APK: (un apk è una specializzazione di .jar)

Resources.arsc: file binario contenente la tabella che mappa le risorse

Classes.dex tutti i .class sono convertiti in DEX e unificati in un solo file.

AndroidManifest.xml

Res/* file delle risorse

META-INF/* contenente i certificati pubblici (certificati RSA/SHA1)

Un file apk quindi è un archivio contenente tutti i componenti di una applicazione: è autodescrittivo grazie ai manifest, è compatto grazie alla compressione e affidabile grazie ad un meccanismo di firma digitale.

Una volta caricata in memoria l'applicazione è distinta in componenti, il glusso di lavoro dell'utente è fatto di componenti appartenenti ad applicazioni diverse eseguiti in processi diversi. Per questo Android incoraggia la condivisione sicura fra applicazioni di dati e funzionalità.

NAVIGAZIONE E COMPILAZIONE RISORSE

L'ordine temporale fra un passaggio fra un task e l'altro costruisce uno stack, il tasto back serve a risalire lo stack.

Le risorse sono dati usati dalle applicazioni. Android distingue due casi:

- resources cioè i dati di cui la struttura è nota al framework
- assets (altri dati).

Gli assets sono aggiunti semplicemente all'APK e acceduti come file normali. Durante il processo di build (tramite Gradle) uno dei tool di "appt" (Android asset packaging tool) processa per prima i file XML e li rende in formato binario, poi genera una tabella di corrispondenza fra ID (definiti negli XML) numerici e offset, genera una classe java (R.java) completa e la compila.

Le risorse semplici sono contenute nei file XML e sono definite come tipo name="pippo"

Le risorse complesse sono identificate da tipo/basename (sono gli stessi file XML ad esempio /res/menu/laymenu.xml)

L'accesso alle risorse può avvenire conoscendo il suo package, il suo tipo o il suo nome.

In un file XML tramite @[package]timo/nome mentre nella parte java [package.]R.tipo.nome.

R.tipo.nome è l'id numerico della risorsa, può essere usato con metodi come setContentView(id).

Per ogni risorsa Android consente di definire quella di default e quelle alternative. Le risorse alternative si possono definire tramite l'uso di qualificatori, nella directory res si creano delle directory qualificate contenenti i file necessari (come immagini o icone).

Ci sono qualificatori per SIM e rete, per lingua e regione, direzione del testo, dimensione schermo ecc.. è importante il qualificatore per la versione dell'OS con formato –v

Esempio pratico: la directory ha nome res/drawable-es-v14/icon.png significa che verrà caricata icon.png nella situazione in cui l'utente è spagnolo e ha come OS Ice Cream Sandwich.

A run-time il resource manager individua per ogni risorsa quale fra le tante usare, ogni volta che la configurazione corrente cambia il sistema riavvia l'activity corrente con il nuovo insieme di risorse.

In particolare, un'Activity può:

Ignorare le modifiche: viene chiusa e riavviata (con le nuove risorse) dal sistema

Salvare e ripristinare: L'Activity non viene chiusa, ma viene chiamato un suo metodo passando i dettagli della nuova situazione

Gestire manualmente il cambiamento: Si dichiara un oggetto che viene preparato dalla "vecchia" Activity e passato alla "nuova", che lo usa per ripristinare uno stato transiente.

E' possibile richiedere tramite manifest che l'app abbia una precisa configurazione all'avvio. Ad esempio se nel manifest è richiesta esplicitamente che il dispositivo abbia un touch screen (android:reqTouchScreen=["undefined" | "notouch" | "stylus" | "finger"] />, altrimenti il dispositivo non è supportato (meglio evitare).

COMPONENTI DI UNA APPLICAZIONE

Le applicazioni android non sono un blocco monolitico ma un insieme di componenti cooperanti:

- Activity: attività atomica dell'utente, concretizzata da una schermata, può essere composta da vari fragment.
- Service: attività del sistema o dell'app invisibile all'utente. Eseguita in background, non interagisce con l'utente.
- Content Provider: componente che pubblica contenuti, viene utilizzato da altre applicazioni
- Broadcast Receiver: componente che ascolta i messaggi globali, quando riceve dei messaggi esegue del codice specifico associato.
- Intent: un messaggio che esprime una intenzione di un utente o di una applicazione affinché qualcosa avvenga. Gli intent possono essere indicizzati a uno specifico

componente oppure emessi in broadcast. Ogni app può definire un filtro che dichiara a quali intent è interessata.

ESECUZIONE APP

Il launcher (che è esso stesso una Activity) lancia la prima Activity dell'app inviando un Intent che lancia il messaggio di avvio. L'Activity chiama `setLayout()` per impostare la sua UI, il sistema chiama le callback in risposta alle azioni dell'utente: lanciare altre activity, recuperare o salvare dati tramite Content Provider, terminare Activity ecc... L'avvio di una Activity può avvenire in due modi:

Explicit intent: creiamo un intent che chiede una activity

Implicit intent: creiamo un intent che chiede una funzione, il sistema cerca quale Activity risponde

Esempio Implicit Intent: ACTION_SEND indica che vogliamo inviare qualcosa a qualcuno, il sistema cerca tutte le app che supportano ACTION_SEND e chiede all'utente quale utilizzare.

Un Intent contiene al suo interno l'azione che vuole eseguire e i dati sui quali dati operare, contiene il componente a cui è indirizzato il messaggio e un bundle Extras (map chiave->valore) di ulteriori campi aggiungibili dal programmatore tramite il metodo `intent.putExtra()`. Come detto precedentemente negli appunti, ogni app definisce un Intent Filter nel AndroidManifest che indica quali messaggi vuole ricevere.

ANDROID MANIFEST

Si tratta del manifesto dell'applicazione, esso include:

- configurazione dell'app, nome, icona, package java ecc...
- informazioni sui permessi necessari (da android 6 in poi i permessi vanno chiesti anche runtime, prima la gestione era statica e venivano chiesti a tempo di installazione)
- elenco dei componenti dell'applicazione
- altri metadati: librerie necessarie...

Android Studio non fornisce una GUI dedicata all'editing del manifesto (cosa che invece viene fornita da Eclipse) però il sistema di build inserisce automaticamente nel manifest le informazioni necessarie per far funzionare Gradle.

ACTIVITY

Le activity sono uno dei componenti di un'applicazione, viene visto come un task dall'utente.

L'utente passa da una activity ad un'altra nel corso del suo task. Si crea quindi uno stack di activity dove l'utente vede solo l'activity in cima allo stack. Quando l'utente preme Home sospende il task corrente, l'intero stack va in background e viene iniziato un nuovo stack. Quando si attiva il task switcher si vedono tutti gli stack, ciascuno rappresentato dall'activity in cima allo stack. Da Android 5.0 è possibile far rappresentare più istanze della stessa activity come task differenti (ad esempio Tab di Google Chrome)

Il comportamento dell'activity in fase di avvio/riavvio può essere controllato dal programmatore tramite i vari FLAG_ACTIVITY_* dell'intent che lancia una activity.

FLAG_ACTIVITY_NEW_TASK: L'activity viene lanciata in un nuovo stack, di cui è l'unico membro
FLAG_ACTIVITY_CLEAR_TOP: Se l'activity esiste nello stack corrente, tutte quelle sopra di essa vengono chiuse, e l'activity diventa top

FLAG_ACTIVITY_SINGLE_TOP: Se l'activity è già quella top nello stack corrente, non viene lanciata

Nel manifest è presente la prima activity di un app, non esiste un Main ma tutte le activity che accettano un intent MAIN/LAUNCHER sono potenziali Main. Alcune Activity hanno senso solo se possono inviare un risultato di qualche tipo a chi le ha invocate (`startActivityForResult()`).

Per esempio: l'activity di sistema per scegliere un contatto dalla rubrica. Al termine dell'activity lanciata, viene invocato il metodo `onActivityResult()` del chiamante, con argomenti che encapsulano la risposta. Normalmente una activity ha un layout che si setta con `setLayout()`, in altri casi può essere anche formata da fragments cui ogni transazione relativa viene inserita comunque nello stack (con il tasto back fa l'undo della transazione e ritorna all'aspetto precedente dell'Activity). Una activity può auto terminarsi con il metodo `finish()` oppure terminarla dall'esterno con `finishActivity()`.

GESTIONE DELLE RISORSE

il sistema può uccidere una activity per liberare risorse, in particolare Android distingue 3 classi di priorità:

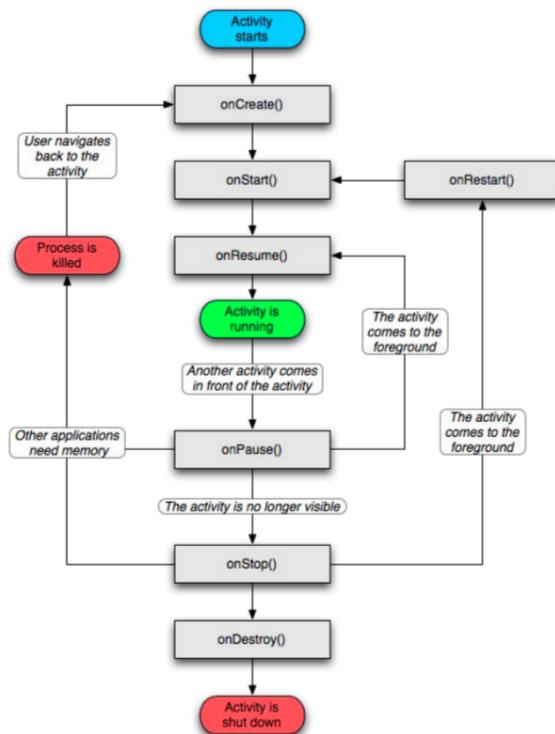
Critica: non vengono mai uccisi

Alta: uccisi in casi disperati

Bassa: uccisi in ordine LRU

Android gestisce dispositivi con memoria limitata, non esiste memoria virtuale, bisogna quindi essere pronti per salvare lo stato e ripristinarlo se necessario.

CICLO DI VITA DI UN ACTIVITY



Avvio `onCreate()` inizializziamo lo stato, `onStart()` sta per essere resa visibile, UI pronta. L'activity cicla tra essere quella in cima allo stack (`onResume()`) ed essere in pausa (`on Pause()`). Solo durante una pausa si può essere uccisi. Se l'activity inoltre non è più visibile `onStop()` l'uccisione diventa più probabile e in caso di utilizzo va riavviata. Una activity in caso di stop può essere distrutta dal sistema, la memoria viene deallocated e il processo viene ucciso. Prima di uccidere l'activity viene chiamata la `onDestroy()` che è l'ultima possibilità di salvare lo stato dell'activity in maniera permanente. E' importante sempre chiamare il super quando si esegue l'override di questi metodi per far sì che il framework continui a fare il suo lavoro.

La Callback `onSaveInstanceState()` viene chiamata dal sistema quando è necessario salvare lo stato dell'activity A BREVE. Ma l'utente può anche terminare l'app con back e in questo caso per il ripristino a lunga si usa `onDestroy()`. A `onSaveInstanceState()` viene passato come parametro un bundle (coppie chiave valore) e dentro troviamo tutti quello che può servire. L'activity può anche essere distrutta ma il bundle sopravvive. Esso sarà poi passato alla `onCreate()`. La classe Activity ha una implementazione di default del salvataggio/ripristino dello stato, scorre il suo layout e salva nel bundle lo stato di tutte le view che a loro volta chiamano il loro `onSaveInstanceState()` (solo le vie che hanno un campo id).

LAYOUT e VIEW

Una UI Android è un albero con foglie di classe View, i nodi intermedi sono di tipo ViewGroup. Un ViewGroup può contenere un numero qualunque di View. La disposizione delle View è regolata da un LayoutManager. (linearlayout, relativelayout, gridlayout ecc...) Una view che gestisce un input è chiamato Widget, quindi una view foglia è un widget. Le librerie di sistema forniscono svariati widget standard ma è sempre possibile scrivere i propri. C'è una corrispondenza (non completa) tra XML e metodi in java per modificare o creare le nostre view / layout.

GESTIONE DELL'INPUT

A runtime esiste un albero di oggetti creato a partire dall'albero XML del layout. Gli oggetti possono ricevere input dall'utente, quando si verifica un evento viene chiamato un handler (`on...Listener()`). Alcune interfacce:

`OnAttachStateChangeListener()`: le View possono essere inserite o rimossa da un albero dinamicamente (a run-time).

`OnClickListener()`: chiamato quando c'è un click sulla View

`onDragListener()`, `onGenericMotionListener()`, `onKeyListener()`, `onLongClickListener()` ecc...

Ogni sottoclasse di View è libera di definire i propri eventi. E' possibile definire nell'XML il nome del metodo da chiamare in risposta ad un evento:

- In layout/....xml

```
<Button ... android:onClick="clicked1" ... />
```

Nel file .java dell'activity

```
public class ... extends Activity {  
    public void onCreate(Bundle b) {  
        ...  
    }  
  
    public void clicked1(View v) {  
        // qui v è il Button che è stato premuto  
    }  
}
```

oppure è possibile associare alla view tramite `setOn....Listener` con argomento una new della classe `On...Listener()`.

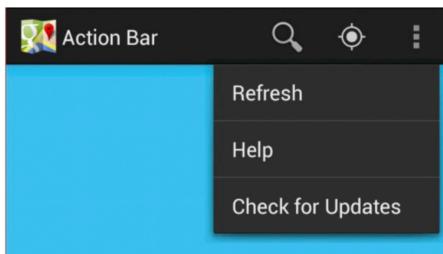
```
if (handlerName != null) {  
    setOnClickListener(new OnClickListener() {  
        private Method mHandler;  
  
        public void onClick(View v) {  
            if (mHandler == null) {  
                try {  
                    mHandler = getContext().getClass().getMethod(handlerName, View.class);  
                } catch (NoSuchMethodException e) { lancia un'eccezione }  
            }  
  
            try { mHandler.invoke(getContext(), View.this); }  
            catch (IllegalAccessException e) { lancia un'eccezione }  
            catch (InvocationTargetException e) { lancia un'eccezione }  
        }  
    });  
}
```

Uso di Reflection!

MENU

Su Android il menu non è un componente grafico ma logico, cioè l'app dichiara quali scelte sono disponibili all'utente. Android utilizza un sistema non-convenzionale per i menu: niente liste gerarchiche con etichette. Il menu "primario" è composto da (solitamente al massimo) 6 caselle con icone e optionalmente testo, se servono più di 6 scelte l'ultima può essere un "Altro..." che mostra una lista di voci più lunga. Una voce può aprire un sottomenu (finestra floating).

Si possono anche avere menù contestuali richiamabili da un long press su una View. Se c'è una ActionBar, il sistema mette le voci più importanti come icone; le altre in un menu "tre puntini".



Il modo più semplice di creare un menù è di usare (come al solito) un file XML in res/menu/ contenente i tag menu e al suo interno vari item che definiscono le voci. Gli <item> possono essere raggruppati logicamente attraverso l'etichetta <group> da cui possono ereditare proprietà "in gruppo".

Come abbiamo già visto per le activity i cicli di vita degli oggetti su android sono controllati dal sistema. La nostra activity dovrà fornire delle callback:

onCreateOptionsMenu() - crea il menu

onPrepareOptionsMenu() - sta per visualizzare il menu e dove posso apportare modifiche al menu come abilitare o disabilitare voci / aggiungere o rimuovere voci / cambiare testo ecc...

onOptionsItemSelected() - reagisce alle selezioni ed è la soluzione più efficiente per gestire la selezione (non si fa nessuna new).

Come regola di stile, è opportuno non configurare troppo il menu, altrimenti l'utente si perde.

MENU CONTESTUALI

Sono l'equivalente Android del "tasto destro", invocati quando si tiene premuto su una View per un tempo lungo. Dipendono dalla particolare View su cui sono invocati mentre i menu visti precedentemente dipendono dall'activity. La creazione di context menu può seguire due strade:

- 1) Si può creare una sottoclasse della View che ci serve, e fare override del suo metodo onCreateContextMenu(ContextMenu cm)
- 2) Si può implementare il metodo onCreateOptionsMenu() dell'activity, e registrare le view che devono invocarlo (Tutte le view non registrate non avranno context menu – registerForContextMenu(View)).

La selezione è gestita come per i menu globali dell'activity in uno dei seguenti modi:

- Registrare un intent direttamente nel MenuItem
- Registrare un MenuItemClickListener nel MenuItem
- Implementare onContextItemSelected(MenuItem menuitem) nell'Activity, come al solito, si discrimina poi in base all'ID di menuitem. (Quest'ultimo è il metodo più efficiente)

Come alternativa ai context menu, è possibile utilizzare una action bar contestuale (CAB).



Viene mostrata solo quando richiesto dal programma. Si sovrappone visivamente alla Action Bar dell'activity, ma è un oggetto separato, si implementa l'interfaccia ActionMode.Callback, si chiama startActionMode() per aprire l'action bar contestuale, per esempio, dentro la onLongClick() di una view.

L'ultimo tipo di menu su android riguarda i pop-up, consente di collegare un pannello menu a qualunque View e il pannello compare subito sotto la View. (Se c'è spazio: altrimenti, prova sopra).

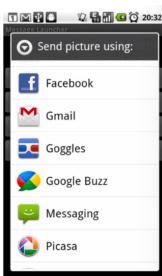
```
public class MainActivity extends ActionBarActivity {  
    private TextView text;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        text=(TextView)findViewById(R.id.text);  
    }  
    public void showPopup(View v){  
        PopupMenu popup = new PopupMenu(this, v);  
        MenuInflater inflater = popup.getMenuInflater();  
        inflater.inflate(R.menu.popup, popup.getMenu());  
        popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener(){  
            @Override  
            public boolean onMenuItemClick(MenuItem item){  
                text.setText("Selezionato: "+item.getTitle());  
                return true;  
            }  
        });  
        popup.show();  
    }  
}
```

Nel layout:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Test popup"  
    android:id="@+id/button"  
    android:layout_centerHorizontal="true"  
    android:layout_marginTop="100dp"  
    android:onClick="showPopup"/>
```

Reagiamo alla selezione con il solito listener

CREAZIONE MENU DINAMICA IN BASE A INTENT



È possibile chiedere al sistema di riempire un nostro menu con tutte le azioni offerte da altri componenti del sistema su dati da noi forniti. E' diverso dal chiedere chi sono i componenti del sistema che possono completare un'azione che noi indichiamo. Si usa il processo di Intent resolution generico, il menu costruito avrà tutti gli item associati con l'Intent "giusto" per far partire il receiver corrispondente.

SCRIVERE UNA PROPRIA VIEW

Gli elementi dell'interfaccia utente sono tutte sottoclassi di view, per realizzare widget custom è sufficiente estendere view o una delle sue sottoclassi. La onMeasure() serve per scegliere le dimensioni, i parametri passati al metodo sono i requisiti, il metodo deve poi fornire la risposta chiamando il metodo setMeasuredDimension(int w, int h), tipicamente si chiama la super.onMeasure() e poi si aggiusta il risultato ottenuto. La nostra sottoclasse dovrà rispondere all'invocazione di onMeasure() specificando la sua dimensione "preferita", nel rispetto dei vincoli. In genere il contenitore effettuerà clipping.

Il metodo onDraw() viene invocato quando la view deve disegnarsi (nello spazio che è stato negoziato dalla onMeasure()), alla onDraw() viene passato un Canvas, quest'ultimo è una superficie di disegno virtuale, supporta le classiche primitive grafiche, clipping e matrici di trasformazione. Il Canvas ovviamente è implementato su una pipeline di rendering, quindi effettua varie operazioni come rendering, rasterizzazione, shading, clipping ecc... prima di restituire la bitmap disegnata). Il nostro codice può disegnare sul Canvas usando una o più Paint, un oggetto della classe Paint rappresenta una specifica completa del modo con cui una primitiva grafica deve essere disegnata.

Andiamo a vedere i costruttori di una View:

`View(Context c)`: costruttore di base che associa la View al suo contesto

`View(Context c, AttributeSet attr)`: costruttore che viene chiamato quando la View viene creata a partire dalla specifica XML in un file di layout

`View(Context c, AttributeSet attr, int stile)`: costruttore che applica anche uno stile identificato dal suo ID di risorsa

Notare come ogni costruttore abbia bisogno del contesto passato come parametro, perché non si può avere una View senza il suo contesto.

Una vista ha delle proprietà come grafica (colore) e contenuto (testo visualizzato) quando queste proprietà cambiano bisogna chiamare la `invalidate()` che invalida la view e il sistema chiamerà la `onDraw()` nuovamente. Molte custom View avranno degli attributi specifici, è quindi sempre buono implementare dei metodi getters e setters e consentire l'accesso a programma oppure in xml si possono definire dei nodi con `<declare-stylable>` definisce i nomi e i tipi degli attributi di una particolare custom view.

SCRIVERE UN PROPRIO LAYOUT

Quando i LayoutManager non soddisfano le nostre necessità si può creare un layout custom consentendoci di disporre i figli View secondo criteri personalizzati. Un layout deve fare override di un solo metodo:

`void onLayout(boolean changed, int left, int top, int right, int bottom).`

Quando il sistema grafico chiama il nostro metodo `onLayout()`, ci sta chiedendo di disporre i figli in maniera tale che tutto il gruppo occupi lo spazio definito da left, top, right, bottom. Molto spesso, si dovrà anche fare override di `onMeasure()` perché le dimensioni del gruppo dipenderanno dalle dimensioni dei figli e dalla disposizione degli stessi.

STILI E TEMI

Sia gli stili che i temi sono coppie attributo-valore, consentono di assegnare ad un nome una configurazione e applicarla riferendosi al nome. Uno stile si applica a una vista e vale solo per quella vista, un tema si applica a una vista, una activity o una intera app e vale per tutte le viste figlie. Un tema può definire anche attributo che si applicano alle finestre e non alle viste.

(`/res/values/styles.xml`). Per applicare uno stile, è sufficiente dichiarare l'attributo `style` nel file di layout XML, riferendo lo stile definito ad esempio `style="@style/light"`. Analogamente, per applicare un tema si usa l'attributo `android:theme="..."` in `AndroidManifest`. Android definisce numerosi stili e temi selezionabili come Temi di default Theme.WithActionBar, Theme.Light ecc... Con Android 5 si raccomanda di usare lo stile Material Design più info sul sito material.io

LISTVIEW E DATA ADAPTER

Uno dei componenti più utilizzati in una interfaccia grafica su android è la lista scrollabile, ogni elemento della lista è a sua volta una view. Quindi la ListView è un ViewGroup pur non essendo un layout. Se le ListView sono statiche si definisce un array di risorse in res/values, si imposta l'attributo android:entries del tag con un riferimento alla risorsa array.

- Res/values/arrays.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="lully">
        <item>1. Jubilate Deo (29 agosto 1660)</item>
        <item>2. Miserere (23 marzo 1663)</item>
        <item>3. Benedictus (1663 o 1664)</item>
        <item>4. O lachrymae (1664?)</item>
        <item>5. Plaudite laetare Gallia (24 marzo 1668)</item>
        <item>6. Te Deum (9 settembre 1677)</item>
        <item>7. De profundis (maggio 1683)</item>
        <item>8. Dies Irae (1 settembre 1683)</item>
        <item>9. Quare fremuerunt (19 aprile 1685)</item>
        <item>10. Domine salvum fac regem (1685?)</item>
        <item>11. Notus in Judea (1685 o 1686)</item>
        <item>12. Exaudiat te Domine (1687)</item>
    </string-array>
</resources>
```

- Res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:entries="@array/lully"/>
```

- Activity (solita solfa)

```
public class ListViewTestActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main); } }
```

Spesso però i dati sono generati dal programma oppure estratti da un database, o ancora ottenuti da un servizio web. In questi casi si accoppia la ListView con un Adapter. Un Adapter ottiene i dati "grezzi" per una entry, costruisce una view contentente i dati, fornisce la view alla listview cui Adapter è associato e notifica agli observer quando i dati cambiano. Ci sono diverse classi Adapter che traducono i dati da diverse fonti:

- ArrayAdapter: traduce i dati da un array prende come parametri il contesto, l'ID del layout XML da usare, ID della testview e l'array.
- Cursor Adapter: adatta i dati di una query SQL
- ResourceCursorAdapter: Adatta i risultati di un cursor con un layout da risorsa XML
- SimpleCursorAdapter: adatta i dati di un cursor mappando nomi di colonna a ID di nodi textview o Imageview
- SimpleAdapter: Usa una ArrayList<Map> una riga oper entry, una chiave nella map per ogni campo della riga.

Uno degli scopi dell'adapter è quello di evitare di tenere tutti i dati in memoria. La Listview e l'Adapter collaborano perché vengano tenuti in memoria e visualizzati solo i dati che servono in un dato momento. Quindi è SBAGLIATO leggere i dati da un DB e caricarli in un array e poi agganciare l'array a un ArrayAdapter. E' CORRETTO chiedere di volta in volta al DB i dati che servono.

Per riconoscere il click su un elemento si implementa l'interfaccia OnItemClickListener, lo si associa alla lista con setOnItemClickListener() e si aspetta che venga chiamato onItemClick().

Android fornisce una sottoclassificazione di Activity specializzata per contenere ListView chiamata ListActivity. Il layout di default contiene due view:

La ListView, con id "@+id/list"

Opzionalmente, una view per il caso di lista vuota, con id "@+id/empty".

È anche possibile usare setContentView() per sostituire un proprio layout a quello di default.

Metodi di Adapter:

public View getView(int position, View convertView, ViewGroup parent): deve restituire una View che rappresenta l'oggetto in posizione position, la View verrà inserita come figlia di parent, se possibile, deve modificare convertView in modo che essa rappresenti l'oggetto, e restituirla. Altrimenti (meno efficiente), può allocare e restituire una nuova View.

RECYCLEVIEW

ListView implementa un comportamento utile in generale, cioè l'adattamento delle View con dati dinamici riciclando le View per evitare new e risparmiare memoria. Però questo comportamento è strettamente accoppiato con la gestione a lista. RecyclerView fornisce:

- Un LayoutManager per decidere come disporre gli elementi (linear, grid e staggered grid)
- Un Adapter per recuperare i dati corrispondenti a un dato indice da mostrare (si estende ViewHolder che è un contenitore per le View cachate) cioè l'operazione di binding per inserire i dati nelle viste.
- Decorazioni per evidenziare una vista
- Animazioni per l'aggiunta e la rimozione di elementi

Il processo di binding consiste nel modificare una View in modo da mostrare i dati corrispondenti a un dato indice, effettuato da due metodi dell'Adapter:

onCreateViewHolder(ViewGroup parent, int tipovista), chiamato quando la RecyclerView ha bisogno di creare un nuovo ViewHolder da inserire nel parent dato

onBindViewHolder(VH holder, int indice), chiamato quando la RecyclerView vuole inserire i dati di indice dato nel VH dato

WEBVIEW

Una webview è una view che incapsula un web browser. Fornisce tutte le funzioni base per caricare una URL e visualizzarla all'utente. Fornisce anche funzioni più specifiche per intercettare funzioni particolari: esecuzione Javascript, gestione cookie... E' possibile anche eseguire il browser esternamente lanciando un Intent. La WebView standard è del tutto adeguata per presentare contenuti in HTML, richiede però ulteriore lavoro per regolare impostazioni più di dettaglio come elementi di UI custom "intorno" alla WebView. La WebView esegue nativamente il codice

Javascript presente nella pagina ma è possibile anche passare una URL nella forma

javascript:funzione(). È anche possibile effettuare il binding fra oggetti Java (dell'App) e oggetti Javascript (nella pagina). Questa tecnica è usata da alcuni ambienti di sviluppo cross-platform basati su tecnologie web, creando così le WebApp. Si scrive una web app "tradizionale", client-side HTML5, CSS3, Javascript Jquery, Bootstrap, node.js, ... La si esegue dentro una WebView "arricchita", l'ambiente aggiunge una serie di oggetti predefiniti, scritti in Java, così è possibile chiamare da Javascript i metodi delle varie classi di Android che non sono emulabili da HTML5, tutta la grafica è renderizzata sul DOM.

DRAWABLE

Drawable è la superclasse degli elementi che possono essere disegnati, esistono molte sottoclassi specializzate ed è possibile definire le proprie ereditando (bitmapdrawable, clipdrawable, colordrawable ecc..). Un drawable può essere definito in un file XML come risorsa e poi caricato dal programma con un normale accesso alle risorse (resources.getDrawable()) e usato per disegnare su un canvas tramite objectdrawable.draw().

- Esempio (definisce uno ShapeDrawable):

res/drawable/gradient_box.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <gradient
        android:startColor="#FFFF0000"
        android:endColor="#80FF00FF"
        android:angle="45"/>
    <padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
        android:bottom="7dp" />
    <corners android:radius="8dp" />
</shape>
```

Una immagine .jpg, .png salvata tra le risorse è una bitmapdrawable la si recupera con resource.getDrawable().

Una immagine ninepatch (tipo vettoriale con caratteristiche specifiche come cornice extra di 1 pixel e stretching arbitrario) è di tipo NinePatchDrawable.

AnimationDrawable mi permette di definire animazioni.

Le icone in android hanno regole di design specifiche, si usa un tool fornito da Google chiamato Asset Studio.

NOTIFICHE ALL'UTENTE

Android prevede tre forme di notifica all'utente:

1. Toast e Snackbar: brevi pop-up che appaiono sopra l'activity e scompaiono automaticamente dopo poco tempo
 - Toast: semplici e veloci, poco intrusivi. Si instanziano con `Toast.makeText(context, text, duration).show();` (si possono definire toast custom tramite XML)
 - Snackbar: La Snackbar è una versione più moderna dei Toast (è consigliato usare la Snackbar anziché i toast su OS recenti). A differenza dei toast, le snackbar sono Widget. Spesso inseriti in un CoordinatorLayout (per avere effetti di animazione complessi)
`Snackbar sb = Snackbar.make(myCoordinatorLayout, R.string.sb, 500).show();`
2. Status Bar: icone nella barra di stato in alto. Le notifiche su barra di stato sono generalmente utilizzate per segnalare eventi in background. Su Android, possono essere generate da Activity (in foreground) o da Service (in background). Le notifiche hanno un ciclo di vita distinto da quello del componente che le ha generate, rimangono attive finché l'utente (o chi le ha generate) non le cancella. Anche se nel frattempo l'Activity che le ha create è morta! Possono anche essere animate, o fornire informazioni di progresso (progress download ad esempio) (pre lollipop la progress bar va animata a mano, post lollipop si usa il metodo `setProgress()`)
3. Dialog: finestre pop-up tradizionali con possibilità di interazione dell'utente. (vedremo dopo più approfonditamente)

Notification: oggetti della classe che rappresentano la singola notifica. Le singole notifiche possono anche fornire layout personalizzati e componenti UI interattivi. Le notifiche hanno cambiato aspetto ad ogni versione di Android – meglio non fare assunzioni.

NotificationManager è un servizio di sistema che gestisce le notifiche, fa visualizzare le notifiche, gestisce lo swipe-down per i dettagli e inizia l'azione richiesta quando l'utente ci clicca sopra.

CREARE UNA NOTIFICA (pre Lollipop)

Le singole notifiche possono anche fornire layout personalizzati.

Per creare l'oggetto: `Notification notification = new Notification(icon, tickerText, when);`

icon – ID di risorsa dell'icona da mostrare

tickerText – testo che scorrerà brevemente sulla barra di stato nel momento in cui viene emessa la notifica

when – istante della notifica, spessissimo: `System.currentTimeMillis()`

CREARE UNA NOTIFICA (post Lollipop)

Da Android Lollipop in poi le notifiche hanno molti più metadati che le collegano al contesto mentale dell'utente, come la persona (o contatto) di riferimento, riferimenti all'App che l'ha prodotta, notifiche su più device e inoltre è stata introdotta una classificazione delle informazioni.

Le informazioni che possono essere visualizzate nella notifica possono essere classificate come:

- Sensibili: da non mostrare sul lock screen
- Compatte: da mostrare nel notification drawer con informazioni parziali
- Estese: da mostrare nel notification drawer con la notifica espansa

La classe NotificationCompat è una factory per creare notifiche compatibili con i vari sistemi. Ci serve un riferimento al NotificationManager di sistema:

```
NotificationManager nm = getSystemService(Context.NOTIFICATION_SERVICE);
```

Poi affidiamo la notifica al manager:

```
nm.notify(id, notifica); (id è un nostro intero che serve a identificare la notifica).
```

Spesso la notifica va modificata/aggiornata anche dopo che è stata emessa. Ad esempio si vogliono dare più informazioni quando l'utente "apre" la barra delle notifiche per avere dettagli o ancora se l'utente seleziona una notifica potremmo voler compiere qualche azione...

Come sempre in Android, se abbiamo l'intenzione di fare qualcosa, la esprimiamo tramite un Intent.

NOTIFICHE COMPLESSE (pre Lollipop)

Il metodo setLatestEventInfo() consente di aggiornare (e completare) i dati di una notifica.

```
Context context = getApplicationContext();
CharSequence titolo = "Titolo (breve)";
CharSequence testo = "Testo (anche lungo)";
Intent i = new Intent(this, MyClass.class);
PendingIntent pi = PendingIntent.getActivity(this, 0, i, 0);

notification.setLatestEventInfo(context, titolo, testo, pi);
```

Intent esplicito

Un PendingIntent è un oggetto che custodisce un Intent pronto a essere spedito in qualche punto del futuro. L'Intent pendente deve essere completamente inizializzato nel momento in cui il PendingIntent viene usato. Per aggiornare quindi una notifica già esistente:

Si chiama setLatestEventInfo() della notifica con i nuovi parametri. Si chiama notify() del NotificationManager passando lo stesso id della notifica precedente.

Ad esempio arriva una nuova email e si aggiorna il conteggio mail "nuove" e si aggiorna l'orario dell'ultimo arrivo. Si può accedere anche brutalmente ai campi es: notification.sound = sound o notification.ledARGB = color.

NOTIFICHE COMPLESSE (post Lollipop)

Una volta ottenuta la Factory NotificationCompat è possibile impostare svariate varianti:

addPerson(), setColor(), setPriority ecc.. E' molto comune che una notifica includa una progress bar, pre Lollipop la progress bar va aggiornata a mano in un thread async (altrimenti bloccherebbe l'app), post Lollipop basta usare setProgress() sul builder e aggiornare la notifica con una notify sul notification manager.

NOTIFICHE HEADS-UP

Quando un'applicazione è in modalità schermo intero sovrascrive anche la barra delle notifiche, quindi esse non sarebbero più visibili. Per questo il sistema le trasforma in notifiche heads-up come una finestra indipendente.

NOTIFICHE ANDROID OREO

Da Android Oreo, sia il comportamento, che le API per le notifiche sono state modificate introducendo i Notification Channels: definiscono "canali" specializzati per certi tipi di notifiche. E' stata introdotta la possibilità di Snoozing, cioè consente all'utente di posporre una notifica per un

certo tempo o di Dismissal, cioè la cancellazione ad da parte dell'utente è ora distinta da quella effettuata via programma. Anche l'aspetto grafico è cambiato, si possono impostare colori e stili grafici distinti senza produrre un'intera view.

NOTIFICATION CHANNELS

Una app può inviare una notifica a uno specifico canale, ogni canale definisce:

- Priorità
- Segnale sonoro, luminoso (LED), vibrazione
- Se le notifiche vadano mostrate sul lock screen
- Se le notifiche vadano mostrate in modalità Do Not Disturb

```
// Creiamo il canale
String id = "RAI1"; // ID del canale
String nome = "Servizio pubblico"; // Label per l'utente
int pri = NotificationManager.IMPORTANCE_LOW; // Priorità
NotificationChannel nch = new NotificationChannel(id, nome, pri);

// Configuriamo il canale
nch.enableLights(true);
nch.setLightColor(Color.GREEN);
nch.enableVibration(true);
nch.setVibrationPattern(new long[]{100, 200});

// Registriamo il canale con il Notification Manager
nm.createNotificationChannel(nch);
```

Per inviare notifiche a un canale particolare, basta configurare il suo builder in maniera corrispondente:

I canali sono creati da una App e vengono registrati tramite il Notification Manager. L'utente (non la app) definisce le proprietà precedentemente definite.

```
Notification n = new Notification.Builder(this)
    .setContentTitle(...)
    .setContentText(...)
    .setSmallIcon(...)
    .setChannel(CHANNEL_ID)
    .setTimeout(System.currentTimeMillis() + delay)
    .build();

nm.notify(id, n);
```

NOTIFICHE IN ANDROID Q

In Android Q, Google ha aggiunto un servizio rimpiazzabile di gestione delle notifiche ancora più semplice, un NotificationAssistantService può ricevere le notifiche poste dalle applicazioni ma anche proporre degli aggiustamenti che ne modificano alcune caratteristiche: come aggiungere Smart Reply predefinite, cambiare priorità, implementare funzioni di snoozing, ecc...

DIALOG

Un Dialog è parte dell'interfaccia utente di una activity ed è sempre collegato al suo contesto, Android mette a disposizione alcuni tipi di dialog già pronti (ovviamente è possibile crearne di propri):

AlertDialog: richieste di decisione

ProgressDialog: stato di avanzamento di un task

DatePicker e TimePicker: input date e orari

Mentre una notifica Toast “galleggia” su un'Activity e non interferisce con essa, e una notifica su status bar è completamente fuori da ogni Activity, un Dialog interagisce con la sua Activity.

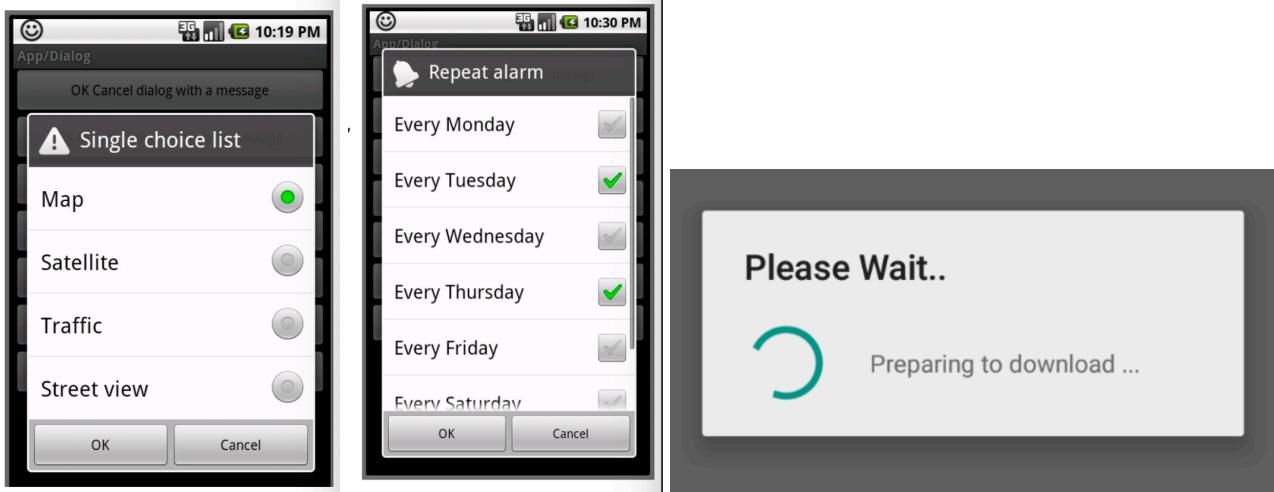
L'Activity chiede al sistema di aprire un suo dialogo (passando un int) con showDialog(id), se è la prima volta che viene richiesto il particolare dialogo, il sistema chiama onCreateDialog(id) dell'Activity, che costruisce e restituisce un Dialog (Il sistema “conserva” il Dialog restituito per usi futuri). Prima di aprire il Dialog, il sistema chiama onPrepareDialog(id, dialog) dell'Activity dove si possono cambiare i contenuti del Dialog.

È possibile creare Dialog custom non usando il Dialog Builder di default, ma si istanziando direttamente il Dialog e si imposta il suo layout (definito in xml) con setContentView(). Da Android 3.0 in poi è consigliato usare un DialogFragment.

Il tipo più semplice (e più generale) è l'AlertDialog e comprende, optionalmente: un titolo, un'icona, un testo e da uno a tre pulsanti. La creazione effettiva viene svolta da un DialogBuilder.

```
protected Dialog onCreateDialog(int id) {
    switch (id) {
        case DIALOG_YES_NO_MESSAGE:
            return new AlertDialog.Builder(this)
                .setIcon(R.drawable.alert_dialog_icon)
                .setTitle(R.string.alert_dialog_two_buttons_title)
                .setPositiveButton(R.string.alert_dialog_ok, new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int whichButton) {
                        /* cose da fare se l'utente ha cliccato OK */
                    }
                })
                .setNegativeButton(R.string.alert_dialog_cancel, new
                    DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int whichButton) {
                            /* cose da fare se l'utente ha cliccato Cancel */
                        }
                    })
                .create();
    }
}
```

Ci sono svariati tipi di Dialog tra cui i Dialog Single choice, MultiChoice e DialogProgress contenente uno spinner circolare di caricamento (quest'ultimo sconsigliato da lollipop in poi poiché blocca l'applicazione quindi ha senso generarlo solo in presenza di più thread).



FRAGMENT

I fragment sono porzioni di UI riciclabili, possono adattarsi a schermi di varia dimensione ma hanno un loro ciclo di vita distinto da quelle delle activity ma integrato con esso (cioè se l'activity termina, termina anche il fragment). Hanno anche un loro salvataggio dello stato (analogo a quello delle activity: onSaveInstanceState() con passaggio del bundle). I fragment sono inseriti in un layoutXML con il tag <fragment> oppure è possibile usare un FragmentManager. Android fornisce sotto-classi specializzate

DialogFragment: implementare Dialog con Fragment

ListFragment: analogo a ListActivity, (Cursor ecc.)

PreferenceFragment: analogo a PreferenceActivity

WebViewFragment: frammento per WebView

MapFragment: mappe di Google Maps

È possibile creare una sottoclassificazione di Fragment facendo obbligatoriamente override del metodo onCreateView() che deve restituire la View corrispondente al Fragment. A livello di programma si gestiscono istanze di Fragment, ogni frammento può essere visualizzato in activity diverse o momenti diversi per questo è fondamentale distinguere lo stato del frammento dallo stato dell'activity. Quando si aggiungono e rimuovono Fragment dinamicamente a una Activity, è necessario usare le transazioni: garantiscono che le operazioni vengano fatte in modo atomico così da non mandare eventi a fragment in corso di rimozione. Quando si aggiungono e rimuovono

Fragment dinamicamente a una Activity, è opzionale usare le **transizioni** (grafiche): Si occupano di rendere esteticamente gradevole l'operazione di sostituzione.

Le **transazioni** si occupano anche di inserire le variazioni di UI ottenute rimpiazzando i frammenti nello stack dei task, quindi se l'utente fa back non ritorna all'Activity precedente ma si toglie il Fragment. È possibile associare una **transizione grafica** (predefinite del sistema o custom-file xml) alla **transazione**. Per specificare la **transizione** da usare, si chiamano metodi della **transazione**

setTransition(int style): usa l'animazione standard

setCustomAnimations(int in, int out): Indica id risorse di tipo animazione da utilizzare per l'ingresso e l'uscita.

setCustomAnimations(int in, int out, int stackin, int stackout): specifica animazioni diverse per l'in/out semplice e l'in/out da stack

DIALOGFRAGMENT

Come abbiamo visto, Activity, Fragment e Dialog hanno ciascuno il proprio ciclo di vita, distinto da (ma integrato con) quello degli altri (solitamente Activity), coordinare questi cicli di vita può essere complicato. Per questo motivo, da Android 3.0 in poi è raccomandato l'uso di DialogFragment al posto di Dialog. DialogFragment è una sottoclasse di Fragment specializzata per mostrare come proprio contenuto un Dialog. La maggior parte delle interazioni col Dialog avviene attraverso metodi di DialogFragment: show(), dismiss(), setStyle(), setShowsDialog()... E' anche possibile accedere al Dialog contenuto nel fragment con getDialog().

SALVATAGGIO TEMPORANEO DELLO STATO

Il ciclo di vita di un'Activity prevede casi in cui l'istanza della vostra classe può essere eliminata dalla memoria, in questi casi è necessario salvare lo stato transiente di un'Activity, in modo da poterlo ripristinare più tardi, quando il sistema istanzierà una nuova copia dell'Activity. Android invoca onSaveInstanceState() quando vuole fare un "commit" dello stato. onSaveInstanceState() salva lo stato in un Bundle. Se necessario, onCreate()/onRestoreInstanceState() ripristina lo stato dal Bundle.

BUNDLE

Il bundle è una mappa chiave-valore, dove chiave è una stringa e valore è un Parcelable (La classe Parcel e l'interfaccia Parcelable sono usate come meccanismo di IPC (inter process communication, serve per comunicare dati tra processi diversi) in Android. I valori Parcelable non sono piena serializzazione ma è molto più efficiente. Il Bundle è un meccanismo generico per passare valori, è possibile istanziare un proprio Bundle, inserire dei valori, e poi metterlo come Extra in un Intent che viene spedito ad altri.

- Salvataggio

```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    // Save away the original text, so we still have it if the activity  
    // needs to be killed while paused.  
    outState.putString(ORIGINAL_CONTENT, mOriginalContent);  
}
```

- Ripristino

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    ...  
    // If an instance of this activity had previously stopped, we can  
    // get the original text it started with.  
    if (savedInstanceState != null) {  
        mOriginalContent = savedInstanceState.getString(ORIGINAL_CONTENT);  
    }  
}
```

Il Bundle preparato da onSaveInstanceState() è mantenuto dal sistema solo se si prevede di far ripartire la particolare istanza dell'Activity. Ovvero, se essa viene scaricata per mancanza di

memoria. NON viene chiamato `onSaveInstanceState()` se l'Activity è terminata con un `Back` o con un `finish()`. Il bundle è mantenuto in maniera non permanente cioè viene salvato in RAM, in caso di riavvio ovviamente si perde tutto. (Ma nulla esclude che il sistema prima o poi decida di tenerlo in NVRAM). Nel caso in cui dei Fragment siano attivi in una Activity, la `onSaveInstanceState()` salva nel bundle anche l'istanza del Fragment.

SHARED PREFERENCES

Per memorizzare dati in maniera permanente si possono utilizzare varie tecniche. Un caso frequente è quello in cui si vogliono memorizzare preferenze e impostazioni dell'utente.

Ma anche per memorizzare propri dati dell'app come statistiche di utilizzo o timestamp dell'ultimo aggiornamento. Android mette a disposizione la classe `SharedPreferences`, rappresenta una mappa chiave valore simile al `Bundle` ma con alcune differenze: le preferenze non sono memorizzate nella RAM ma in maniera permanente. I valori possono essere solo tipi base `String`, il sistema gestisce l'aggiornamento atomico, è disponibile un sistema di notifiche quando le preferenze cambiano. Le preferenze vengono salvate su un file, se ne possono avere svariate per questo è possibile dare un nome ad un insieme di preferenze (Path sul file system: `/data/data/package/shared_prefs/nome.xml`). Un'altra differenza con i bundle è il fatto che le scritture sono transazionali nel senso che prima viene fatto un insieme di aggiornamenti poi si effettua il commit che le scrive tutte insieme. Ciò garantisce consistenza (non può capitare che alcuni campi si aggiornino e altri no) e coalescing (chi è in attesa di notifiche viene notificato una sola volta sola per l'intero insieme di modifiche). Le modifiche alle preferenze vengono eseguite tramite un Editor che offre metodi del tipo “`putTipo(chiave, valore)`”. Per copiare la tabella temporanea sulle preferenze sono disponibili due metodi dell'editor:

- `commit()` : aggiunge la tabella dell'editor a quella delle preferenze, e salva immediatamente su disco (Sicura, in caso di errore restituisce un codice d'errore, meno efficiente)
- `apply()` : aggiunge la tabella dell'editor a quella delle preferenze in memoria, e schedula la scrittura asincrona del risultato su disco (Meno sicura, non verifica gli errori, più efficiente)

E' possibile ottenere un oggetto di tipo `SharedPreferences` dotato di nome (pubblico) tramite `getSharedPreferences(nome,modo)` dove modo può essere `MODE_PRIVATE`, `MODE_WORLD_READABLE`, `MODE_WORLD_WRITEABLE` ecc..

Oppure un oggetto privato con `getPreferences(modo)`

Di default per un dato Context si usa `PreferenceManager.getDefaultSharedPreferences(ctx)`

Esempio:

- Scrittura

```
SharedPreferences pref=getPreferences(MODE_PRIVATE);
Editor editor=pref.edit();
editor.putString(K_LOGIN, login);
editor.putString(K_PWD, password);
editor.commit();
```

- Lettura

```
SharedPreferences pref=getPreferences(MODE_PRIVATE);
login=pref.getString(K_LOGIN, "guest");
password=pref.getString(K_PWD,"123456");
```

Le `SharedPreferences` possono essere condivise tra più Activity, ciascuna di esse può modificarle (tanto il commit è atomico) e le altre possono voler essere informate del cambiamento. Per questo

esistono le notifiche sulle preferenze si usa il solito meccanismo dei Listener, il Listener deve implementare l'interfaccia OnSharedPreferencesChangeListener che ha un unico metodo: onSharedPreferenceChanged(prefs, chiave). Per rimuovere un listener ovviamente unregisterOnSharedPreferenceChangeListener(listener).

I metodi che abbiamo visto sono relativamente semplici e piuttosto comodi per salvare un po' di dati generici. Tuttavia, quando le preferenze rappresentano davvero preferenze dell'utente, e devono essere editabili, le cose si complicano. Serve un'Activity per fornire l'interfaccia grafica dove vanno gestiti tutti gli eventi relativi alla modifica dei parametri, Android fornisce un framework ad-hoc!

PREFERENCE SCREEN

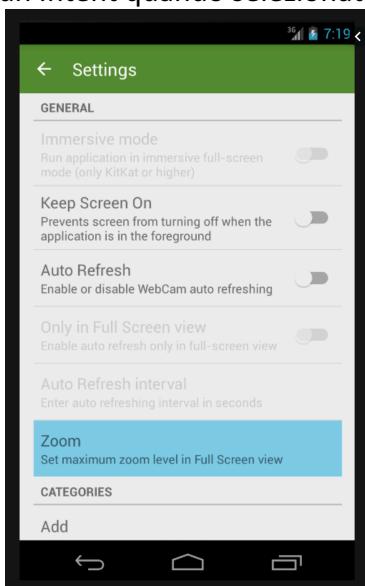
Si può costruire una GUI corrispondente alle impostazioni che l'utente deve modificare. Conviene allora definire le preferenze in maniera dichiarativa e lasciar fare al sistema sfruttando un file XML. L'editing, l'aggiornamento, le notifiche, l'undo (con Back) ecc. sono gestiti autonomamente. Il file che descrive le preferenze va in res/xml non è un layout, anche se il sistema ne deriva un layout.

L'elemento radice è <PreferenceScreen> All'interno, può contenere:

<PreferenceScreen> - struttura gerarchica

<PreferenceCategory> - raggruppa opzioni correlate

È possibile avere "preferenze" che non consentono di impostare alcunché, ma si limitano a inviare un Intent quando selezionate. Per esempio: un "About" messo fra le preferenze.



La classe PreferenceActivity si occupa di leggere il nostro prefs.xml, generare al volo un layout, interagire con l'utente e salvare le impostazioni alla fine delle modifiche. L'activity parte, presenta l'interfaccia dei settings all'utente, come bonus ulteriore si adatta automaticamente al tema corrente. L'uso di riferimenti a risorse nel file XML rende anche il tutto facilmente localizzabile. Anche la gestione dei Dialog "interni" per cambiare i parametri è completamente invisibile, i valori scelti però verranno scritti alla fine nelle preferenze.

PREFERENCE FRAGMENT

Da Android 3.0 esiste la versione Fragment della Preference Activity. Ha diversi vantaggi come l'essere inclusa in schermate più complesse, può supportare diversi layout e maggiore integrazione con transazione e transizioni. Lo svantaggio è la quantità supplementare di codice da scrivere.

Definizione del Fragment:

```
public class TestPreferenceFrag extends PreferenceFragment {...}
```

Aggiunta a una Activity:

```
getFragmentManager().beginTransaction() .replace( ... , new TestPreferenceFrag()) .commit();
```

ACCESSO AL FILE SYSTEM

Android è uno strato “sopra” Linux, il File System è quindi da esso gestito (File, directory, hard/soft-link, diritti, proprietario, ecc.). La gestione dei file di Android è dunque uno strato (interfaccia Java) “sopra” il File System vero, accedere direttamente ai file su Android è raro. Si usano più spesso le preferenze o i database, oppure, in sola lettura, le risorse (raw) e gli assets. È ovviamente disponibile la gestione dei file standard di Java tramite le classi dei package java.io/java.nio e i vari wrapper come Buffered(input/output)stream ecc...

Tuttavia, Android presenta delle peculiarità: l'utente non siete voi, ma la vostra App o la coppia (user,app). Ogni app è un utente diverso, i file sono mutuamente segregati in directory distinte e di default i file sono “privati” all'app (ma possono essere resi pubblici).

I dispositivi distinguono memoria “interna” (al telefono) e memoria “esterna” (scheda SD e simili). L'ambiente definisce posti standard in cui memorizzare vari tipi di dato condivisi (Musica, video, ebook, suonerie, foto, podcast...). Ogni App ha una base directory in memoria interna e una in memoria esterna (se c'è). Queste directory vengono cancellate se l'app viene disinstallata. L'App inoltre ha una cache directory per i file temporanei, il sistema la cancella se ha bisogno di spazio. Il sistema fornisce una directory condivisa per dati pubblici a cui anche altre app possono accedere. Il context offre metodi di utilità per accedere ai file nella base dir interna:

```
FileOutputStream openFileOutput(nome, modo)
```

```
FileInputStream openFileInput(nome)
```

...

e altri metodi per gestire le directory

```
File getFilesDir(): restituisce il path alla base directory
```

```
File getDir(nome, modo): apre o crea una sottodirectory
```

```
File getCacheDir(): restituisce il path alla cache directory
```

Prima di accedere alla memoria esterna è bene controllare che sia presente, usando

```
String getExternalStorageState() i file sono accessibili se risponde con
```

Environment.MEDIA_MOUNTED o Environment.MEDIA_MOUNTED_READ_ONLY sono disponibili in sola lettura altrimenti MEDIA_REMOVED vuol dire che non è presente. Usando File getExternalFilesDir(tipo) viene restituita la directory in cui l'App dovrebbe salvare i file del tipo indicato, tipicamente /Android/data/package/files/... Tuttavia, la memoria esterna è condivisa, non ci sono diritti né protezioni: è solo una convenzione!

Un'applicazione che voglia esplicitamente condividere file (tipicamente, media), può salvarli nella directory condivisa di sistema File getExternalStoragePublicDirectory(tipo). Questi file non vengono cancellati quando l'applicazione viene disinstallata, se l'app ha bisogno di una cache più ampia, può utilizzare la memoria esterna chiamando File getExternalCacheDir(). La cache esterna verrà svuotata alla disinstallazione ma non se la SD è piena. (la gestione è a carico del programmatore).

INTERNAL VS EXTERNAL STORAGE

La memoria interna è sempre disponibile, i dati salvati sono accessibili solo dall'app, quando l'app è disinstallata tutti i dati vengono cancellati.

La memoria esterna non è sempre disponibile, perché l'utente può inserire o rimuoverla. Tutti possono accedere ai file della SD, al massimo sta all'app crittografare i dati che salva sulla SD in modo tale che possano essere letti solo da essa. Quando l'utente disinstalla l'app i file vengono rimossi solo nel caso in cui l'utente li abbia salvati nella cartella `getExternalFilesDir()`.

È sempre possibile accedere a file tramite tutte le API che accettano URI, usando lo schema `file:///.Uri.fromFile()` per ottenere l'URI dal file.

Il FileProvider è un particolare ContentProvider fornito dal sistema dedicato alla gestione dei file. Può essere configurato (tramite file XML) per fornire alle applicazioni una visione “virtuale” di File System.

ACCESSO AL DATABASE

Android incorpora una versione di SQLite per l'utilizzo e creazione di database di uso generale. È implementato come una libreria dinamica (.so) efficiente con piccoli database. Solo l'app può accedere ai suoi database. Si possono esporre i dati ad altri tramite Content Provider.

La classe `SQLiteDatabase` rappresenta un singolo DB, identificato tramite il nome del file .db.

Esistono due pattern tipici di accesso a DB:

- Usare `SQLiteDatabase` e i metodi relativi per creare e modificare il DB “a mano”
- Creare una sottoclasse di `SQLiteOpenHelper` per innestare sui suoi metodi di ciclo di vita le operazioni sul DB in maniera “assistita”

SQLITEDATABASE

Per aprire o creare un database si possono usare vari metodi statici di `SQLiteDatabase` `openDatabase(path, factory, flags)`, path è il pathname del DB, factory è la classe da invocare per creare i Cursor, flags indica il modo di apertura, bitmask fra:

`OPEN_READWRITE`

`OPEN_READONLY`

`CREATE_IF_NECESSARY`

`NO_LOCALIZED_COLLATORS`

Frequentemente: `SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(path,null);`

Più raramente: `SQLiteDatabase db = SQLiteDatabase.create(null)` che è una alternativa dove si crea il database in memoria (non salvato in un file!) per memorizzare in maniera temporanea dati su cui sia utile operare in maniera relazionale. Un DB in memoria è molto veloce, ma viene cancellato al momento della `close()`!

SQLITEOPENHELPER

Il secondo pattern tipico per l'uso di DB prevede che si crei una sottoclasse di `SQLiteOpenHelper`.

Questa classe fornisce:

- Un costruttore che associa l'helper a un DB
- Metodi di utilità per l'apertura del DB
- Event handler per gestire creazione o upgrade del DB
- Gestione automatica delle transazioni su ogni operazione

Costruttore: SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)

Factory può essere null, il numero di versione (monotono crescente) serve a decidere quando occorre fare l'upgrade di un DB. Per esempio, perché è arrivata una nuova versione dell'applicazione oppure si è aggiunta una colonna nel DB. Il costruttore non apre di default il DB, si tratta quindi di un pattern Lazy. L'Helper offre due metodi di utilità per aprire il DB:

getReadableDatabase(): apre in sola lettura

getWritableDatabase(): apre in lettura/scrittura

Se il DB non esiste, viene invocato l'handler onCreate() dell'Helper.

Se il DB esiste, si legge il suo numero di versione. In particolare:

Se il numero di versione del DB è uguale a quello nel costruttore dell'Helper, il DB è pronto per l'uso.

Se il numero di versione del DB è minore di quello nel costruttore dell'Helper, viene invocato onUpgrade()

Se il numero di versione del DB è maggiore di quello nel costruttore dell'Helper, viene invocato onDowngrade()

A questo punto, viene invocato onOpen(). Se non ci sono stati errori, il DB viene restituito al chiamante.

La sottoclasse deve implementare la onCreate() (dove si decide lo schema) e la onUpgrade() (viene eseguita la drop table if exist e si crea la tabella nuova da zero chiamando la onCreate()).

Riassumendo, si crea una sottoclasse di SQLiteOpenHelper per ogni database che usiamo. Nella onCreate() dell'Activity, si costruisce un'istanza della sottoclasse con opportuni parametri. Quando è necessario accedere al DB si invocano i metodi getReadable/WritableDatabase() sulla istanza della sottoclasse.

OPERAZIONI SUL DATABASE

Una volta ottenuto (in qualunque modo) un db, possiamo eseguire le consuete operazioni SQL.

Il metodo più generale è db.execSQL(sql), esegue i comandi SQL passati (come stringa), il comando passato può contenere qualunque comando, purché non debba restituire nulla (il metodo è void).

In particolare, può eseguire CREATE TABLE e simili. NON può eseguire SELECT, può eseguire UPDATE, ma non restituire il numero di record modificati.

Il costo di compilazione di ogni istruzione SQL non è (affatto) trascurabile, per questo SQLite fornisce una modalità alternativa, in cui anziché passare una istruzione SQL, si invocano specifici metodi:

delete(tabella, where, args)

insert(tabella, nullcolumn, valori)

```
ContentValues cv = new ContentValues();
```

```
cv.put("nome", "Seminari Est");  
cv.put("edificio", "Marzotto C");
```

```
db.insert("Aule", null, cv);
```

replace(tabella, nullcolumn, valori)

update(tabella, valori, where, args)

ContentValues offre varianti overloaded del metodo put() che accettano valori di tutti i tipi base. Si occupano loro della conversione da tipi Java a tipi SQL. La Select invece viene utilizzata per ottenere i dati che possono servirci per riempire, ad esempio, una ListView. La Select viene eseguita attraverso un Cursor che ci permette poi in seguito di scorrere i risultati.

Eseguire una SELECT come statement SQL: Cursor rawQuery(sql, args)

Eseguire la SELECT a programma: Cursor query(distinct, tabella, colonne, selezione, args, groupby, having, orderby, limit)

L'oggetto Cursor consente di scorrere i risultati di una query. Concettualmente, è un puntatore al record corrente all'interno di una tabella di risultati. Appena ottenuto un Cursor, esso è posizionato sul primo record (se c'è) dei risultati. Ci sono metodi di spostamento come moveToPosition(indice), moveToFirst() e metodi per leggere la posizione getPosition().

L'idea era che i Cursor avessero un ciclo di vita complesso con la possibilità di sospendere lo scorrimento di un result set, ripetere la query per avere dati aggiornati, e riprendere lo scorrimento. In realtà la cosa non ha mai funzionato bene, la pratica raccomandata è quella di creare semplicemente un nuovo Cursor quando serve.

androidx.ROOM

Fornisce uno strato di astrazione ulteriore rispetto a SQLite, si gestiscono oggetti, non record o tabelle. (relational-object bridging & persistance library)

```
@Entity
public class User {
    @PrimaryKey
    private int uid;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;
    ...
}

@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE "
            + "first_name LIKE :first AND "
            + "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);
    ...
}

@Database(entities = {User.class}, version = 1)
public abstract class MyDB extends RoomDatabase {
    public abstract UserDao userDao();
}
```

```
MyDB db =
    Room.databaseBuilder(appctx,
        MyDB.class, "db-name").build();
```

CONTENT PROVIDER

Abbiamo visto che le applicazioni Android possono utilizzare file e DB SQL per la memorizzazione. In generale, si vuole poter condividere i dati fra più applicazioni indipendenti. In maniera universale ma controllata è possibile accedere ai dati resi disponibili da altri (ContentResolver) e rendere i propri dati accessibili agli altri (ContentProvider). Il metodo query del ContentResolver implica eseguire una query molto costosa, ma eseguirla nel thread della UI, la classe CursorLoader viene in aiuto caricando asincronamente i risultati di una query (vedremo dopo).

I dati dei ContentProvider sono le URI, esse hanno un formato noto:

content://media/internals/images

content: schema

media: identifica il provider (authority)

internals/images: è il path

La classe android.net.Uri fornisce metodi per costruire e convertire URI. Ad esempio Uri Uri.parse(String s) data una stringa s, costruisce l'Uri relativa. La classe Uri rappresenta una URI immutable, è efficiente, ma poco flessibile. A differenza della classe Uri.Builder che è un costruttore di URI progettato per manipolare URL mutable, sono presenti più metodi per alterare le componenti e alla fine, si chiama il suo metodo build() per ottenere una Uri. ContentUris fornisce invece metodi statici di utilità per manipolare le URI con schema content://. Se si dispone dei giusti permessi, è possibile anche modificare righe esistenti, o aggiungerne di nuove. Si tratta sempre di operazioni richieste al ContentProvider starà a lui implementarle.

Per inserire un nuovo record, si crea un oggetto ContentValues (mappa colonne-valori) e si invoca il metodo insert() del ContentResolver.

```
ContentValues cv = new ContentValues();
cv.put("Name", "Vincenzo");
cv.put("Surname", "Gervasi");
cv.put("Age", 48);
Uri newrow = cr.insert(uri, cv);
// il campo _ID è aggiunto automaticamente
```

Uri della tabella
Uri del nuovo record

per modificare invece uno o più record esistenti si usa il metodo update() del ContentResolver.
Analogamente si possono cancellare uno o più record.

Chi offre un Content Provider può richiedere dei permessi nel suo manifesto, chi invece accede a un Content Provider deve usare i permessi nel suo manifesto (se richiesti).

Lo scopo è duplice: garantire che l'utente approvi i permessi in fase di installazione e garantire che chi usa un ContentProvider “lo conosca” a fondo. (Il nome del permesso fa un po' da “password”)

DEFINIRE UN CONTENT PROVIDER

```
<provider android:name="ArcobalenoProvider"
    android:authorities="it.unipi.di.sam.arcobaleno"
    android:exported="true"
    android:enabled="true"
    android:description="@string/abdesc">
</provider>

public class ArcobalenoProvider extends ContentProvider {
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) { /* ... */ }
    @Override
    public String getType(Uri uri) { /* ... */ }
    @Override
    public Uri insert(Uri uri, ContentValues values) { /* ... */ }
    @Override
    public boolean onCreate() { /* ... */ }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
    { /* ... */ }
    @Override
    public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) { /* ... */ }
}
```

Come al solito, usiamo il nome del package come prefisso → unicità

Qualunque applicazione può definire un ContentProvider (identificato da una authority) per offrire accesso ai propri dati. Un ContentProvider è un componente top-level dell'applicazione (come le Activity), ha una sua sezione in AndroidManifest.xml. Per iniziare si crea una sottoclasse di ContentProvider. Il nostro ContentProvider avrà il suo ciclo di vita distinto da quello dell'Activity! Normalmente viene gestito automaticamente dal sistema, viene istanziato quando un ContentResolver deve gestire un'URI la cui authority corrisponde alla nostra, viene chiuso quando non è più necessario. Il metodo onCreate() deve creare o aprire il DB

corrispondente. Compiere molte operazioni con un Content Provider out-of-process può essere costoso (vengono compiute molte operazioni di serializzazione dei dati e IPC). È possibile anche qui fare coalescing cioè si descrivono le operazioni da fare “in blocco”, si spedisce l'intero pacchetto di operazioni con una sola comunicazione IPC (l'operazione è anche atomica!) ContentProviderOperation fornisce metodi che restituiscono un Builder specializzato per le varie operazioni possibili: newInsert(), newUpdate(), newDelete(), ecc. Ciascun Builder fornisce metodi per specificare ulteriori argomenti withValue() / withValues(), withSelection(), ecc.

Ovviamente, non vogliamo spargere stringhe con authority per i provider, path per le tabelle, nomi di colonna ecc. in giro per il nostro codice. La pratica raccomandata consiste nel fornire una classe contratto che definisce costanti simboliche per tutti questi valori.

FILE PROVIDER

A volte, si vorrebbero usare delle API che si aspettano di accedere a un Content Provider, anche se i dati veri e proprio sono ospitati come file (sul file system). FileProvider è un Content Provider implementato nella libreria di supporto (da v4 in poi), come tutti i provider, va dichiarato nel Manifest. La struttura delle directory su file system associate al file provider sono definite in un file XML.

```
<paths>
    <files-path path="images/" name="myimages" />
</paths>
```

La "tabella" myimages sarà mappata su: «basedir»/it.unipi.di.sam.myapp/files/images/
Usando file:// come URI si applicano ovviamente dei permessi del file system di Linux. Usando invece contents://file-provider/ si applicano i permessi di Android. È possibile creare un Intent che fa riferimento a un URI sul file provider su cui si impostano i flag permessi di lettura/scrittura. Quando si passa l'Intent a un'altra Activity quest'ultima avrà il permesso di accedere al file identificato dall'URI.

Un metodo alternativo (ma meno bello) consiste nell'assegnare esplicitamente i permessi di accesso a una URI per uno specifico package (di altra app) tramite i metodi di Context:
grantUriPermission(package, URI, flags),
revokeUriPermission(package, URI, flags)

ESECUZIONE CONCORRENTE – MULTITHREADING

Staticamente un pezzo di codice appartiene a un metodo, che appartiene a una classe, che appartiene ad un package che appartiene a una applicazione. Dinamicamente un pezzo di codice è eseguito da un thread che appartiene a un processo che appartiene ad una applicazione. Su Android un processo ha spazio degli indirizzi isolato.

In Java la classe thread rappresenta il thread, non il codice da eseguire. Mentre l'interfaccia Runnable presenta il codice da eseguire non il thread che lo esegue. L'interfaccia Runnable rappresenta un task, la classe thread rappresenta un flusso di esecuzione. L'oggetto Thread rappresenta un thread della JVM ma non lo è, finché non viene avviato, un thread è semplicemente un oggetto Java in memoria. L'avvio avviene chiamando il metodo start() del thread, parte un nuovo thread che esegue il metodo run(). La sincronizzazione di thread avviene attraverso l'uso di monitor, ogni oggetto java ha un monitor associato, obj.wait() sospende fino alla notify (riparte) o interrupt. Prima di poter invocare però wait o notify il thread deve acquisire il monitor di obj, questo può essere fatto tramite synchronized. Il comando di un metodo synchronized (espr) { blocco } prova ad acquisire il monitor dell'oggetto denotato dall'espressione, si sospende se il monitor è occupato, rilascia il monitor all'uscita dal blocco. I costrutti synchronized offrono un modo per realizzare la mutua esclusione e per serializzare l'accesso da parte di diversi thread.

DUE REGOLE D'ORO

1. Mai usare thread della UI per operazioni lunghe
2. Mai usare un thread diverso dal thread UI per aggiornare la UI.

PROBLEMA: Come posso fare se serve una operazione lunga che deve aggiornare la UI?

Ad esempio un accesso a DB, accesso alla rete, calcoli "pesanti". Creare nuovi Thread mi aiuta per la regola #1, non per la #2.

```
class MyTask extends AsyncTask<Integer, Float, Void> {

    @Override
    protected void doInBackground(Integer... params) {
        int limit=params[0], sleep = params[1];
        for (int i=0; i<limit && !isCancelled(); i++) {
            try {
                Thread.sleep(sleep);
            } catch (InterruptedException e) { ; }
            publishProgress((float)i/limit);
        }
        publishProgress(1.0f);
        return null;
    }

    @Override
    protected void onProgressUpdate(Float... p) {
        progressbar.setProgress((int)(p[0]*100));
    }
}
```

Il caso più comune è quando il thread UI deve far partire un task (lungo), durante lo svolgimento deve aggiornare la UI, in fine deve fornire il risultato alla UI. Per questo particolare caso, è molto comodo usare la classe (astratta e generica) AsyncTask. Come in altri casi, dovremo creare una nostra sottoclasse e fare override di metodi. Un task deve implementare doInBackground() essendo un metodo astratto. AsyncTask è una classe generica, può operare su tipi diversi, al momento dell'istanziazione si specificano i tipi effettivi tra <>.

AsyncTask<Tipo Argomenti del task, Tipo del progress report, Tipo del risultato>

- Metodi da implementare
 - Ciclo naturale
 - void `onPreExecute()`
 - **R** `doInBackground(A...)`
 - void `onProgressUpdate(P...)`
 - void `onPostExecute(R)`
 - Cancellazione anticipata
 - void `onCancelled(R)`
- Metodi da chiamare dall'esterno
 - **Costruttore**
 - AsyncTask `execute(A...)`
 - `cancel(boolean interrupt)`
 - **R** `get()`
 - AsyncTask.Status `getStatus()`
- Metodi da chiamare dagli on...
 - ()
- void `publishProgress(P...)`
- boolean `isCancelled()`

Metodi che sono eseguiti dal thread UI
• Devono essere veloci, ma possono interagire con la UI
Metodi che sono eseguiti dal thread in background
• Possono essere lenti, ma non devono interagire con la UI (o invocare altre funzioni del toolkit)

AsyncTask è solo una classe di utilità per organizzare I thread in uno schema frequente. Ci sono comunque primitive per fare comunicare I thread non-UI con il thread UI in altre strutture. In qualche caso, Android offre garanzie specifiche sul modello di threading che riducono la necessità di usare synchronized. Nota bene: se mai il thread UI dovesse incontrare un synchronized, sarebbe bloccato finché il thread che attualmente possiede il monitor non ha finito!

La classe Activity offre il metodo void runOnUiThread(Runnable r), esso può essere chiamato da un thread non-UI, il Runnable (task) sarà eseguito dal thread UI dell'Activity (in qualche momento del futuro). Utile, per esempio, per aggiornamenti “volanti” di una progress bar o rinfrescare una ListView man mano che arrivano dati.

La classe View invece offre:

```
void post(Runnable r)  
void postDelayed(Runnable r, long millis)
```

Possono essere chiamati da un thread non-UI, il Runnable sarà eseguito dal thread UI della View a cui questa View appartiene. Non può essere invocato se la View non è inserita nel layout di una Activity!

Se classi e metodi di utilità messi a disposizione dalla libreria non bastano, si può scendere al livello sottostante:

Handler: gestisce la MessageQueue di un thread

Message: busta per un Bundle

MessageQueue: coda di Message

Looper: classe che offre un ciclo lettura-dispatch da MessageQueue

Ogni Activity ha un Looper eseguito dal thread UI, i vari post() accodano nella MessageQueue del Looper dell'Activity un Message con la specifica dell'operazione richiesta (come Parcelable) (siamo al livello più basso di android). Quando abbiamo detto che “dopo la richiesta il sistema, con suo comodo, in qualche punto del futuro, farà la tale operazione” in realtà si intendeva che la richiesta crea un Message che descrive l'operazione, lo passa all'Handler, che lo accoda nella MessageQueue, da cui verrà estratto da un Looper che eseguirà l'operazione.

IMAGEDOWNLOADER MULTITHREAD

Vogliamo creare una classe in grado di scaricare una immagine dal web e inserire la bitmap corrispondente in una ImageView. Lo scaricamento avverrà su thread separato, l'inserimento nella view su thread UI. Usiamo l'AsyncTask nel metodo doInBackground andiamo a scaricare la bitmap usando la URL che ci è stata passata come parametro nell'AsyncTask, nel metodo onPostExecute() andiamo a prendere l'ImageView e associamo la bitmap.

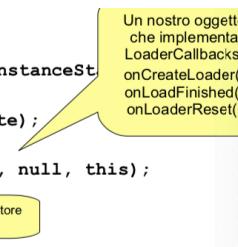
WEAK REFERENCES

Normalmente tenere un riferimento (puntatore) a un oggetto impedisce al Garbage Collector di disalocarlo. Il risultato è che se l'Activity è stata chiusa, la memoria occupata non può essere liberata, nel nostro esempio andiamo a creare nell'AsyncTask un nuovo riferimento all'ImageView. Per evitare questa situazione usiamo i Weak References. Un Weak Reference si comporta come un riferimento ma non impedisce al Garbage Collector di disaloccare la memoria (sempre se non ci sono altri riferimenti non weak) in caso di disallocazione al get() restituisce null.

CURSORI ASINCRONI

La classe astratta Loader specifica un protocollo generico per il caricamento asincrono di dati. Si tratta di componenti attivi. A Loader attivo, quando i dati sottostanti vengono aggiornati, il Loader deve trasferire gli aggiornamenti al suo utente e il trasferimento è spesso (ma non necessariamente) asincrono. I Loader sono tipicamente associati ai Cursor. Ogni Activity ha associato un LoaderManager (quindi il ciclo di vita dei Loader dipende da quello dell'Activity), l'inizializzazione del LoaderManager viene eseguita nella onCreate() dell'Activity, si occupa anche di ripristinare lo stato salvato dei Loader.

```
@Override  
public void onCreate(Bundle savedInstanceState)  
{  
    super.onCreate(savedInstanceState);  
    ...  
    getLoaderManager().initLoader(0, null, this);  
    ...  
}
```



L'ID numerico identifica univocamente un Loader ma se ne possono avere più di uno nella stessa Activity. Il comportamento tipico: è che se non esiste già un Loader con quell'ID, viene creato e inizializzato passando il secondo parametro al suo costruttore, altrimenti se invece esiste già, viene

ripristinato il suo stato precedente (se c'è) dal Bundle. Per re-inizializzare un Loader già esistente si usa il metodo restartLoader(). Il nostro onCreateLoader() deve restituire un Loader inizializzato, il cursorloader poi, ad esempio, fa partire una query asincrona che darà risultato un Cursor nella onLoadFinished(). Quando un Loader sta per essere chiuso o ripristinato il LoaderManager invoca la onLoaderReset().

DOWNLOAD HTTP UTILIZZANDO DOWNLOADMANAGER

Android offre un servizio di sistema per il download di risorse via HTTP, il servizio è ovviamente asincrono, gestisce lo scaricamento, la notifica di progress e segnala il completamento all'applicazione. Come per tutti i servizi di sistema, si può ottenere con:

```
DownloadManager dm = getSystemService(DOWNLOAD_SERVICE)  
e una volta ottenuto il servizio, si possono accedere download con  
long id = dm.enqueue(request)
```

al termine si viene avvisati da un broadcast ACTION_DOWNLOAD_COMPLETE. L'app dovrà registrare quindi un BroadcastReceiver (vedremo dopo).

La classe DownloadManager.Request incapsula una richiesta di scaricamento, contiene tutti i dettagli necessari fra cui L'URI da scaricare, dove mettere il file locale (per default, spazio condiviso), se far visualizzare una notifica oppure no durante il download associandogli un titolo e una descrizione. Possono essere associate restrizioni sulle reti da usare: solo in wi-fi, LTE o anche in roaming ecc.

Immaginiamo di avere un pulsante “download”...

```
public void onClick(View view) {
    dm = (DownloadManager) getSystemService(DOWNLOAD_SERVICE);
    u = Uri.parse(bella_url); // Uri u
    r = new Request(u); // Request r
    r.setTitle("Una bella Url");
    r.setDescription("probabilmente fantastica!");
    r.setAllowedOverRoaming(false); // non così fantastica!
    r.setVisibleInDownloadsUi(true);
    id = dm.enqueue(r); // long id
}
```

Registriamo un Receiver

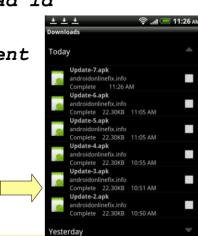
```
registerReceiver(receiver,
    new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE)
);
```

In questo caso ha senso usare la registrazione dinamica: registriamo il Receiver quando avviamo un download e lo de-registriamo al completamento. In altri casi potrebbe essere meglio registrarlo staticamente (tramite AndroidManifest.xml).

Il nostro Receiver sarà:

```
public void onReceive(Context context, Intent i) {
    String act = i.getAction();
    if (DownloadManager.ACTION_DOWNLOAD_COMPLETE.equals(act)) {
        long id = i.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, 0);
        // qui gestiamo il completamento del download id
    } else {
        // chissà, potremmo aver ricevuto altri intent
    }
}
```

Altri Intent utili
ACTION_NOTIFICATION_CLICKED – lanciato se l'utente seleziona la notifica del download
ACTION_VIEW_DOWNLOADS – da lanciare (via startActivity()) per attivare l'interfaccia del download manager stesso (“Downloads”)



Il modo più semplice per accedere al file appena scaricato è di invocare alcuni metodi del DownloadManager stesso. Ad esempio ParcelFileDescriptor pfd = dm.openDownloadedFile(id); restituisce un file descriptor (serializzabile) sul file scaricato, in sola lettura.

Oppure Uri u = dm.getUriForDownloadedFile(id); restituisce una URI sul file scaricato.

Il DownloadManager mantiene un ContentProvider con le informazioni sui download così che le app possano fare query sul provider per avere informazioni sui file scaricati. La classe DownloadManager.Query fornisce metodi di utilità per le query più frequenti.

ASYNCPLAYER

Molte classi di sistema gestiscono il multithreading completamente al loro interno. AsyncPlayer è una di queste, riproduce file audio, ovviamente in background. L'uso da parte delle applicazioni è banale, se sono richieste capacità più avanzate si può usare l'architettura del MediaPlayer.

BROADCAST RECEIVER

Si tratta di una classe che ha lo scopo di ricevere e rispondere agli Intent inviati in broadcast. Si estende BroadcastReceiver, si pubblica il componente in AndroidManifest.xml con i relativi Intent Filter. I BroadcastReceiver hanno un ciclo di vita semplicissimo, viene instanziato quando se ne ha bisogno, l'Intent viene passato al suo metodo onReceive(), al ritorno da onReceive() l'oggetto viene rilasciato. La onReceive() non ha lunga durata, può però invocare un Service (vedremo come), o lanciare task asincroni (senza risultato). Generalmente, meglio non lanciare un'Activity, ma ci sono eccezioni, es: telefonata in arrivo → parte il dialer. La onReceive() viene eseguita normalmente nel thread della UI, quindi, non lo si può bloccare a lungo. Per sicurezza, il

sistema uccide l'applicazione se la onReceive() dura più di 10 secondi, in realtà, qualunque cosa oltre i 0.5s fa insospettire l'utente. Molti servizi di sistema che inviano delle notifiche broadcast finiscono per essere gestiti da un BroadcastReceiver. E' brutto se arriva un SMS, e parte l'Activity di messaggistica, meglio: arriva un SMS, parte un BroadcastReceiver, il quale posta una notifica nella Notification Area, che poi quando selezionata dall'utente fa partire l'Activity di messaggistica. Esempio di dichiarazione di un BR in Manifest.xml

```
<receiver android:name="CallReceiver">
    <intent-filter>
        <action android:name="android.intent.action.PHONE_STATE">
        </action>
    </intent-filter>
</receiver>
```

- Implementazione di CallReceiver:

```
public class CallReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        if (extras != null) {
            String state = extras.getString(TelephonyManager.EXTRA_STATE);
            if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
                String phoneNumber =
                    extras.getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
                Log.d("CALL", phoneNumber);
            }
        }
    }
}
```

È anche possibile registrare e deregistrare dinamicamente un BroadcastReceiver, in questo modo, l'applicazione riceve broadcast solo quando il receiver è registrato. Tramite i di Context: registerReceiver (BroadcastReceiver receiver, IntentFilter filter) e unregisterReceiver (BroadcastReceiver receiver).

Alcuni Intent inviati in broadcast possono essere definiti "sticky". Un Intent normale raggiunge il ricevitore a cui è destinato, quindi termina. Un Intent sticky rimane invariato in modo da poter avvisare altre app (lanciate in seguito) se necessitano delle stesse informazioni. Quando registri una nuova app che deve conoscere le informazioni o quando viene lanciata un'app inattiva, lo sticky broadcast verrà inviato al destinatario della nuova app. Cioè dopo essere stati inviati in broadcast a tutti i receiver del sistema (il cui IntentFilter corrisponde all'Intent) poi rimangono "vivi". Se successivamente al broadcast si registra dinamicamente un nuovo BroadcastReceiver(di un'altra app) che corrisponde, l'Intent sticky viene inviato normalmente al nuovo receiver (dell'app) e viene anche restituito dalla registerReceiver(). Gli Intent inviati come sticky sono mantenuti dal sistema in una cache, così sono subito disponibili a componenti abilitati in seguito. Per usare gli intent sticky, le app devono avere il permesso BROADCAST_STICKY. (In effetti, sono tipicamente usati (solo) dal sistema). Ad esempio. Il BatteryManager invia un intent sticky per indicare il livello di carica corrente della batteria (e anche lo stato di ricarica o meno). In questo modo, qualunque applicazione può recuperare il più recente Intent sticky.

Nella maggior parte dei casi, le applicazioni saranno interessate a ricevere i broadcast di sistema. Broadcast "privati" di un'app richiedono di conoscere l'app e la struttura/semantica dell'Intent. Comunque, si possono pur sempre inviare propri Intent in broadcast tramite:

sendBroadcast(Intent intent): invia intent a tutti i BroadcastReceiver corrispondenti

startActivity(intent) invia intent a una Activity corrispondente //modo standard

la differenza tra questi due metodi è che l'Intent è lo stesso, i meccanismi di dispatching sono diversi!

La chiamata a sendBroadcast() è asincrona, ritorna immediatamente al chiamante, nel frattempo il sistema, con calma, manda gli Intent.

L'overload sendBroadcast (Intent intent, String receiverPermission) invece invia solo a chi ha dichiarato un certo permesso. Normalmente, i BroadcastReceiver di app diverse vengono eseguiti concorrentemente, ma è possibile chiedere invece l'invio serializzato e ordinato (in base al valore di android:priority del receiver) tramite sendOrderedBroadcast (Intent intent, String receiverPermission). In risposta a un sendOrderedBroadcast(), i Receiver possono anche abortire il broadcast e restituire risultati. Per controllare l'abort anzitempo: get/abortBroadcast().

Per avere dei risultati, si invia l'Intent con:

sendOrderedBroadcast (Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, ...). Si attiva un folding con resultReceiver in fondo. Il processo di folding consiste nel passare a ogni receiver il risultato (cumulato) dei receiver chiamati prima di lui. Ciascun receiver può leggere il risultato dei predecessori, e impostare il proprio. In rari casi, può essere utile inviare Intent in broadcast solo all'interno della propria app. In questo caso, la classe LocalBroadcastManager fornisce alcuni metodi di utilità. (evitando così anche il costo della serializzazione)

ALARM

Fra i molti servizi di sistema di Android, uno si occupa di impostare ed inviare Allarmi. L'invio di un allarme è realizzato tramite l'invio di un Intent (esplicito) al componente che ha registrato l'alarm. Si ottiene il puntatore all'AlarmManager invocando getSystemService()
AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE).

am.set(int type, long triggerAtTime, PendingIntent operation)

Un allarme è definito da un tipo e da un tempo, viene fornito anche il PendingIntent da lanciare quando scatterà l'allarme. Il tipo può essere:

ELAPSED_REALTIME – tempo dal boot

ELAPSED_REALTIME_WAKEUP – tempo dal boot, ma in più sveglia il dispositivo se è in sleep

RTC – real time clock

RTC_WAKEUP – real time clock, ma sveglia il dispositivo se è in sleep

am.setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation) imposta un allarme con ripetizione, verrà inviato l'intent al triggerAtTime, e poi ogni interval (finché non viene cancellato).

am.cancel(PendingIntent operation) cancella tutti gli allarmi registrati con l'Intent passato

Usare gli Alarm ha senso quando si vuole essere svegliati a un dato momento, anche se la vostra applicazione non è in esecuzione al momento! Però per compiti di temporizzazione pura, meglio usare altri costrutti come quelli nativi di Java: Thread.sleep(), System.currentTimeMillis() oppure, usare un Handler con postDelayed().

Gli alarm scattano (ovviamente) anche quando il telefono è in sospensione (sleep). Durante la onReceive() di chi ha ricevuto il PendingIntent, il telefono viene risvegliato e tenuto sveglio ma al ritorno dalla onReceive(), può tornare immediatamente in sleep.

Se la onReceive() ha fatto partire qualche attività asincrona, questa potrebbe fermarsi immediatamente perché il telefono torna in sospensione. Nelle versioni precedenti di Android 4.0 gli allarmi erano sempre esatti, dai successivi target API19 in poi gli allarmi sono per default inesatti, il sistema si sente libero di aggiustarli per minimizzare il consumo di batteria. Le applicazioni però possono controllare la quantità di delay e l'allarme scatterà in qualunque punto della parte concessa. Gli allarmi pendenti vengono annullati con lo shutdown del dispositivo.

Molto poco adatto, per esempio, per una sveglia. La riabilitazione va fatta a mano: si registra staticamente un BroadcastReceiver per l'intent ACTION_BOOT_COMPLETED, richiede il permesso

RECEIVE_BOOT_COMPLETED, ottenuto staticamente. Nella onReceive(), si recuperano i dati delle sveglie (per esempio, da un DB) e si re-impostano gli allarmi.

MODALITA' DOZE

Quando il sistema è in modo Doze non si ha accesso alla rete, vengono ignorati i Wake Lock ma soprattutto gli allarmi dell'AlarmManager possono essere posticipati. Con schermo spento e device stazionario per un certo tempo gli allarmi possono essere ritardati. In modalità Doze non viene eseguito lo scan delle reti wifi. È possibile però chiedere esplicitamente di impostare degli allarmi critici che superano il Doze:

```
am.setAndAllowWhileIdle(int type, long trigger, PendingIntent operation)  
am.setExactAndAllowWhileIdle(int type, long trigger, PendingIntent operation)
```

Usare questi allarmi, ovviamente, peggiora la durata della batteria
... e state sicuri che gli utenti vi scopriranno!

WORK MANAGER

WorkManager è una delle (tante) API di Jetpack, dedicata all'esecuzione affidabile di task asincroni deferrable coordinati e vincolati.

Asincrono = viene eseguito in qualche punto del futuro

Deferrable = può essere rimandato senza danno

Coordinati = si possono esprimere strutture (task in sequenza, in parallelo, ecc.)

Vincolati = per ogni task, si possono specificare condizioni che devono essere verificate per l'esecuzione.

Un task da eseguire è definito da una istanza di Worker, come al solito faremo la extends della classe, con un solo metodo da implementare doWork(). La richiesta di esecuzione di un task (ovvero, di un Worker) è definita da un'istanza di una sottoclasse di WorkRequest. Il sistema fornisce due sottoclassi predefinite:

OneTimeWorkRequest: esegui il task una volta sola

PeriodicWorkRequest: esegui il task ripetutamente.

È possibile accodare più richieste insieme che vengono poi svolte in parallelo, oppure metterle in sequenza. Per finire, è possibile specificare quali vincoli (di sistema) devono essere verificati affinché un lavoro sia eseguibile, i vincoli sono rappresentati come istanze di Constraints.

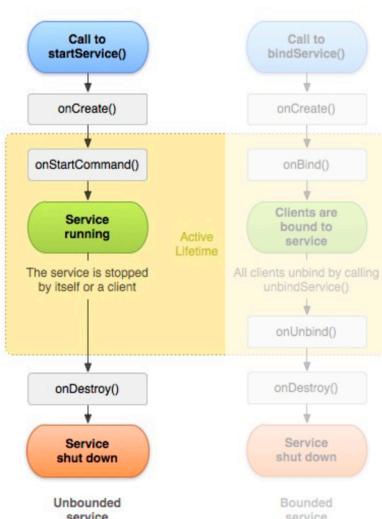
SERVICE

Dopo le Activity, i Content Provider e i Broadcast Receiver, i Services sono il quarto tipo di componente di una applicazione. Un Service è un oggetto che può eseguire del codice senza

disporre di una interfaccia utente, opera sempre in background rispetto alla UI, ma il thread dove viene eseguito è quello della UI. Le Activity di una applicazione possono avviare uno o più dei propri service. Il service ha un proprio ciclo di vita distinto: Cambia a seconda che sia avviato per servire una singola richiesta (**started**), oppure per servire un flusso di richieste da un altro componente (**bound**). Il sistema può sempre uccidere un Service se ha la necessità di memoria.

Un service bound ha sempre priorità almeno uguale a quella dell'Activity che ha chiesto un servizio.

Un service started per essere avviato basta inviargli un Intent ed eseguire una call a startService(intent) chiedendo se il service sia attivo, se è attivo non succede nulla altrimenti viene instanziato



e inizializzato (con la `onCreate`). Poi passerà l'intent a `onStartCommand()`. (i metodi `onCreate` e `onStartCommand()` vengono chiamati sul thread della UI! Di solito, vengono subito creati dei Thread per fare il "lavoro vero", e si torna immediatamente al chiamante.)

Nel caso di Service bound, il sistema tiene traccia di quanti clienti al momento usano, il Service ha un inizio `bindService()` e una fine `unbindService()`. Quando il conteggio raggiunge 0, il Service può essere dismesso. Le connessioni sono permanenti.

ESEMPIO: Immaginiamo di voler scrivere un player musicale, il player avrà un'Activity con la sua interfaccia utente, dovrà ovviamente riprodurre i brani musicali sia quando la GUI è in primo piano sia quando la GUI è coperta da altro, o sono andato nella Home, o proprio uscito. Ovviamente, se rientro nella GUI devo poter controllare il playback!

Manifest

```
<activity android:name=".MainActivity"
    android:label="@string/app_title"
    android:theme="@android:style/Theme.Black.NoTitleBar">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<service android:exported="false" android:name=".MusicService">
    <intent-filter>
        <action android:name="com.example.android.musicplayer.action.TOGGLE_PLAYBACK" />
        <action android:name="com.example.android.musicplayer.action.PLAY" />
        <action android:name="com.example.android.musicplayer.action.PAUSE" />
        <action android:name="com.example.android.musicplayer.action.SKIP" />
        <action android:name="com.example.android.musicplayer.action.REWIND" />
        <action android:name="com.example.android.musicplayer.action.STOP" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.example.android.musicplayer.action.URL" />
        <data android:scheme="http" />
    </intent-filter>
</service>
```

MainActivity

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mPlayButton = (Button) findViewById(R.id.playbutton);
    /* ... */
    mEjectButton = (Button) findViewById(R.id.ejectbutton);

    mPlayButton.setOnClickListener(this); →
    /* ... */
    mEjectButton.setOnClickListener(this); →
}

public void onClick(View target) {
    if (target == mPlayButton)
        startService(new Intent(MusicService.ACTION_PLAY));
    else if (target == mPauseButton)
        startService(new Intent(MusicService.ACTION_PAUSE));
    else if (target == mSkipButton)
        startService(new Intent(MusicService.ACTION_SKIP));
    else if (target == mRewindButton)
        startService(new Intent(MusicService.ACTION_REWIND));
    else if (target == mStopButton)
        startService(new Intent(MusicService.ACTION_STOP));
    else if (target == mEjectButton) {
        showUrlDialog();
    }
}
```

MusicService

```
@Override
public void onCreate() {

    mNotificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    mAudioManager = (AudioManager) getSystemService(AUDIO_SERVICE);

    // Create the retriever and start an asynchronous task that will prepare it.
    mRetriever = new MusicRetriever(getApplicationContext());
    (new PrepareMusicRetrieverTask(mRetriever, this)).execute();

    mDummyAlbumArt = BitmapFactory.decodeResource(getResources(), R.drawable.dummy_album_art);
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    String action = intent.getAction();
    if (action.equals(ACTION_TOGGLE_PLAYBACK)) processTogglePlaybackRequest();
    else if (action.equals(ACTION_PLAY)) processPlayRequest();
    else if (action.equals(ACTION_PAUSE)) processPauseRequest();
    /* ... */
    else if (action.equals(ACTION_URL)) processAddRequest(intent);
    return START_NOT_STICKY; // Means we started the service, but don't want it to restart in case it's killed.
}
```

Il Service è in esecuzione anche se l'Activity è chiusa ma può sempre inviare un Intent per riattivarla, per esempio se la playlist è vuota. (Meglio però farlo via Notification!)

Una volta avviato con startService(), un servizio può essere terminato in tre modi:

Il servizio stesso chiama stopSelf() - uccide se stesso.

Un altro componente chiama stopService(Intent i) passando un Intent che identifica il servizio - viene ucciso da un altro.

Il sistema uccide forzosamente il servizio perché ha bisogno di memoria - uccisione di gruppo.

Le richieste di servizio devono essere terminate nello stesso ordine in cui sono state ricevute.

Terminazione e Riavvio

onStartService() restituisce un valore numerico che indica come gestire i restart forzati

START_STICKY: se il servizio è stato fermato, appena possibile verrà chiamato nuovamente
onStartService() per riavviarlo, passando un Intent null

START_NOT_STICKY: se il servizio è stato fermato, verrà riavviato solo se ci sono chiamate a
onStartService(...,id) pendenti, ovvero non pareggiate da stopSelf(id)

START_REDELIVERY_INTENT: se il servizio è stato fermato, verrà riavviato passando a
onStartService() l'Intent originale

L'argomento flags di onStartCommand() indica la ragione del riavvio

0: non è un riavvio (primo avvio)

START_FLAG_REDELIVERY: si tratta di una ri-consegna dell'Intent. Il sistema aveva ucciso il servizio
prima che il processing fosse concluso, ossia prima della chiamata a stopSelf() a seguito di

START_REDELIVERY_INTENT da onStartCommand()

START_FLAG_RETRY: si tratta di un ri'avvio dopo uccisione forzosa A seguito di START_STICKY da
onStartCommand()

Il metodo startService() restituisce un'istanza di ComponentName. Questo oggetto rappresenta il
componente Service ma può anche rappresentare un'Activity, un BroadcastReceiver o un
ContentProvider (insomma, è una classe generale).

Se startService() restituisce null, l'Intent non è stato consegnato, altrimenti si può usare il
ComponentName per ottenere informazioni sul servizio. Es: getClass(), getClassName(),
getPackageName()

RIASSUNTO SUI SERVICE

Il componente che necessita di un Service prepara un Intent che lo identifichi e aggiunge come
data o extra, tutto il necessario. Il componente chiama startService(intent), il sistema "trova" il
Service giusto e se necessario, lo lancia → onCreate() poi onStartCommand(intent, flag, id).

Il Service serve l'intent, eventualmente invia una risposta al componente, infine il Service chiama
stopSelf(id) oppure rimane indefinitamente in giro, finché qualcuno non chiama stopService().

INTENTSERVICE

Quasi sempre, è utile che un Service usi uno o più thread separati per servire le richieste. Bisogna
ricordare che i metodi del ciclo di vita di un Service sono eseguiti dal thread UI. IntentService è una
sottoclasse di Service che: serve le richieste in un thread separato ed in caso di richieste multiple,
gestisce una coda. (non gestisce richieste in parallelo; se serve il parallelismo, bisogna
implementare da soli il "giusto" meccanismo).

IntentService crea un thread, looper, message queue, handler nella sua onCreate(). Il
ServiceHandler si limita a passare i messaggi (prelevati dalla coda) a un metodo onHandleIntent()
implementato da IntentService. onHandleIntent(Intent intent) è un metodo astratto che bisogna
implementare in una sottoclasse di IntentService.

In definitiva:

Create una sottoclasse di IntentService

Fare overload di onHandleIntent()

Il vostro codice sarà eseguito da un thread separato e le richieste vengono serializzate (non serve synchronized). Quando la coda è vuota, il service (si) termina.

A partire da Oreo (8.0+, target 26+), Android forza alcune “ottimizzazioni di batteria” per i service started: se un service è in qualche modo visibile all’utente, per esempio perché ha emesso una notifica, è considerato in foreground (nel senso UI). In caso contrario, è considerato in background. Se l’app a cui il service appartiene è in foreground non ci sono problemi, se invece l’app a cui il service appartiene è a sua volta in background, il service non viene eseguito (Pur non essendo logicamente terminato).

Sempre da Android 8.0+, ci sono alcune nuove strutture per gestire queste limitazioni:

JobIntentService: come IntentService, ma implementata con Job periodici anziché con Service
startForegroundService(): come startService(), ma consente a un’app in background di lanciare un service in foreground. Il service deve chiamare startForeground(id, Notification) per postare una notifica entro 5 secondi dall’avvio altrimenti, l’app viene uccisa.

SERVICE BOUND

I service started hanno un’interazione limitata con i loro utenti, è possibile in alternativa effettuare un binding. Il service e il componente che lo usa vengono legati in modo più stabile e continuativo. La connessione fra i due rimane finché non viene fatto esplicitamente l’unbound. Il componente può chiamare direttamente metodi del Service in-process, con un IBinder (Interfaccia che definisce i metodi del Service chiamabili dall’esterno) oppure un service cross-process, con AIDL (Android Interface Definition Language, linguaggio definito per IPC).

Se il servizio che vogliamo inizializzare è di tipo “bindato” non viene lanciato con startService(), ma con bindService(intent, connection, flags).

intent: l’Intent che identifica il Service, come prima

connection: un oggetto di classe ServiceConnection che controlla il tempo di vita del binding

flags: precisa la priorità del servizio.

La bindService(intent, conn, flags) causa una chiamata alla onBind(intent) del Service (e forse una onCreate()). Questo metodo è di tipo void e termina subito: il binding è asincrono.

La onBind(intent) restituisce un nostro oggetto binder che implementa l’interfaccia IBinder.

Il binder viene passato alla onServiceConnected() di conn. Da qui in avanti, il chiamante usa i metodi del binder, alla fine, si chiama unbindService(conn).

Abbiamo un rischio sicurezza se usiamo un Intent implicito in bindService(), non possiamo sapere quale Service risponderà, ci aspettiamo di invocare dei metodi che magari il service che ha risposto non ha. Da Android 5.0+, non si può chiamare bindService() con un Intent implicito, viene lanciata un’eccezione.

È perfettamente possibile che un Service offra sia un’interfaccia unbound che una bound
onStartCommand() → interfaccia unbound

onBind() → interfaccia bound

Tuttavia, il ciclo di vita si fa complicato assai. Meglio, in generale, scegliere uno stile e mantenerlo e non mischiare bound con unbound.

RIASSUNTINO SUI VARI TIPI DI SERVICE

Activity: ho una UI

Service: non ho una UI (ma posso avere notifiche)

Unbound: il servizio processa singole richieste

START_STICKY: il servizio può essere ucciso mentre processa una richiesta; in tal caso riattivalo appena possibile (perdendo la richiesta)

START_NOT_STICKY: il servizio può essere ucciso mentre processa una richiesta, in tal caso non riattivarlo fino alla prossima startService()

START_REDELIVER_INTENT: come START_STICKY, ma in più inoltra gli Intent delle richieste non ancora terminate

Bound: il servizio non può essere ucciso mentre è bound

GESTIONE DEI SENSORI

Android implementa un sistema di gestione dei sensori generico chiamato SensorManager, pronto per essere esteso a tipi di sensori diversi.

```
SensorManager sm = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Per recuperare l'elenco di tutti i sensori presenti sul dispositivo:

```
List<Sensor> list = sm.getSensorList(Sensor.TYPE_ALL);
```

Si può avere più di un sensore per tipo, per ottenere il sensore di default per un certo tipo:

```
Sensor sens = sm.getDefaultSensor(tipo);
```

Per ottenere la lista di tutti i sensori di un certo tipo:

```
List<Sensor> list = sm.getSensorList(tipo);
```

Alcuni di questi sensori saranno veri sensori hardware mentre altri saranno sensori software che utilizzano i sensori hardware per ottenere nuove informazioni. Come ad esempio integrando un sensore di accelerazione, si può ottenere un sensore di velocità. Si può controllare la disponibilità di un tipo particolare di sensore necessario a run-time come abbiamo visto con i metodi precedenti. Ma si può anche usare AndroidManifest.xml dove però implica che l'app non sarà installabile su device senza (ad esempio) la bussola

```
<uses-feature  
    android:name="android.hardware.sensor.compass"  
    android:required="true"  
/>
```

Visto che, come risaputo, non siamo noi a chiamare il codice, si registra un listener (indicando anche con che frequenza vogliamo essere chiamati), il SensorManager chiamerà poi i nostri listener, forse con la frequenza richiesta, più o meno... In genere, con frequenza non minore di quella richiesta a causa dell'ottimizzazione, consumo batteria, alarms ecc...

Metodi dell'interfaccia SensorEventListener

L'interfaccia del listener offre metodi per ricevere gli eventi (lettura di valori) e le variazioni di accuracy. Gli eventi sono codificati come SensorEvent e contengono un campo accuracy.

onSensorChanged() cambia il valore letto dal sensore, per esempio, leggo la bussola e il telefono viene ruotato.

onAccuracyChanged() cambia l'accuracy del sensore per esempio, passo dalla localizzazione GPS a quella radio.

È sensato registrare il listener nella onResume() e de-registrarlo nella onPause() della mia activity.

```
boolean success = sm.registerListener(  
    SensorEventListener l,  
    Sensor sens,  
    int rate  
);
```

Registra "l" per essere informato degli eventi relativi a "sens", con una frequenza di circa "rate"

Per deregistrare un listener:

Da uno specifico sensore: sm.unregisterListener(l,sens);

Da tutti i sensori su cui è registrato: sm.unregisterListener(l);

E' importantissimo deregistrare il listener quando non si stanno usando i sensori, altrimenti un processo sta sempre ad aspettare che qualcosa accada, la CPU quindi è sempre attiva e si ha una pessima durata della batteria.

L'accuracy non è un valore di accuratezza riguardo al valore, è una indicazione relativa allo stato del sensore: SENSOR_STATUS_ACCURACY_HIGH/LOW/MEDIUM/UNRELIABLE. Ogni singola lettura di sensore contiene anche l'accuracy che aveva quel sensore in quel determinato momento.

Spesso il metodo onAccuracyChanged non serve ma va comunque eseguito l'override.

Ricapitolando un SensorEvent incapsula:

- Un numero variabile di valori float che rappresentano la lettura del sensore
- Il sensore da cui vengono i valori
- L'accuratezza del sensore al momento della lettura
- L'istante della lettura
- L'interpretazione dei valori dipende dal sensore, se è temperatura saranno gradi Celsius, se è gps saranno latitudine,longitudine e altezza...

I sensori di posizione sono molto complicati, dipendono dalla situazione in cui si trova il telefono, dalla sua rotazione eccetera, La classe SensorManager fornisce un certo numero di metodi e costanti di utilità come getInclination(), getOrientation() e altri.

In generale più precisione e più frequenza implicano più utilizzo di batteria, per lo stesso motivo non è corretto lasciare i listener registrati più del tempo necessario.

Non dedicarsi a operazioni lente nei metodi on...() piuttosto si può memorizzare il valore in una struttura dati e processarlo su un altro thread.

Il SensorEvent passato al listener rimane di proprietà del SensorManager quindi mai tenere dei riferimenti (puntatori di vario genere) all'evento che restano anche dopo il ritorno da onSensorChanged(). Il SensorManager potrebbe usare un pool di SensorEvent per evitare di fare una new per ogni lettura cosicché gli stessi oggetti vengono riciclati più volte.

SERVIZI DI LOCALIZZAZIONE

L'approccio è simile a quello dei sensori, si interpella un servizio di sistema per avere accesso ai vari fornitori di informazione sulla posizione, si registrano dei listener e opzionalmente si lancia un Intent sotto certe condizioni.

Il servizio offerto si chiama LocationManager:

LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

Analogamente a Sensor, esiste una classe LocationProvider che descrive i diversi componenti che possono dare la posizione ad esempio GPS, GPS-A, triangolazione di celle radiomobile, triangolazione di reti wi-fi ecc... Il LocationProvider descrive anche la sua precisione, consumo energetico, ecc...

Per ottenere una lista dei nomi dei vari provider disponibili:

List<String> provs = lm.getAllProviders()

solo i provider attualmente abilitati: List<String> provs = lm.getProviders(enabled)

ottenere un provider indicato per nome: LocationProvider prov = lm.getProvider(name);

Il "miglior" provider in base a certi criteri: LocationProvider prov = lm.getBestProvider(criteria, enabled);

I criteri per selezionare il miglior provider possono essere vari: ACCURACY: COARSE, LOW, HIGH, FINE / POWER: LOW, MEDIUM, HIGH/ ALTITUDE (tutti i criteri sono incapsulati nella classe android.location.Criteria)

È anche possibile indicare esplicitamente un provider desiderato come LocationManager.GPS_PROVIDER o LocationManager.NETWORK_PROVIDER tuttavia, potrebbero esserci dei provider non- standard sul vostro dispositivo.

È possibile registrare un listener che verrà chiamato dopo un intervallo di tempo prefissato oppure dopo che è stata percorsa una distanza prefissata, una sola volta, in base ai dati forniti dal provider o in modo continuo fino alla deregistrazione. Il metodo principale di LocationManager è requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener). Per deregistrare un listener si chiama la removeUpdates(LocationListener listener). Come al solito, è molto opportuno deregistrare i listener nella onPause() e abilitarli nella onResume(). A meno che, naturalmente, non stiate usando un Service o un thread in background per fare rilevazione in continuo...

Molti metodi di localizzazione richiedono parecchi secondi (o minuti!) per ottenere il fix, ad esempio il GPS appena attivato, deve prima trovare i satelliti. In questi casi, si può usare la tecnica seguente:

```
lm.requestLocationUpdates(prov, mint, mind, listener);
Location current=lm.getLastKnownLocation(prov);
...
void onLocationChanged(Location loc) { current=loc; }
...
/* usa current dove serve */
```

Gli oggetti di classe Location contengono sempre latitudine, longitudine, timestamp. Opzionalmente: velocità, direzione, altitudine, accuracy o altre informazioni come getExtras() che contiene al suo interno satellites=n per il GPS.

Un'applicazione può adottare varie strategie per scegliere, fra varie posizioni restituite, quella da usare. Può decidere se usare una locazione più recente, ma meno accurata, oppure una meno recente ma più accurata. Può decidere quanto può essere vecchia al massimo una posizione prima di scartarla del tutto oppure decidere se una posizione è “plausibile” (esempio app navigatore auto). Decidere che frequenza di aggiornamento richiedere, in base al costo energetico del provider oppure un provider che ha perso la posizione ci spostiamo su un altro provider. Per mantenere una posizione corrente affidabile vengono usati più provider insieme. Nel caso in cui invece bisogna georeferenziare un post o una foto scattata, nel momento in cui l'utente inizia la creazione del contenuto l'app inizia l'ascolto della posizione, quando l'utente ha terminato termina anche il listener e viene associata la posizione al post/foto.

L'accesso alle informazioni di posizione è consentito solo alle app che richiedono i relativi permessi nel loro AndroidManifest.xml. È bene spiegare in qualche modo all'utente perché volete tracciare la sua posizione. Importante chiarire quali sono i vostri impegni alla riservatezza dei suoi dati.

Articolo 5.1.2 (garante europeo della privacy)

[...] Il fornitore di un'applicazione in grado di elaborare dati di geolocalizzazione è il responsabile del trattamento dei dati personali derivanti dall'installazione e dall'utilizzo dell'applicazione [...]

Esempio GPS con listener:

- In AndroidManifest.xml

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

- Nella onCreate() dell'Activity

```
current = null; /* una Location, globale */  
lm =  
(LocationManager) getSystemService(Context.LOCATION_SERVICE);  
prov = LocationManager.GPS_PROVIDER;  
aggiorna(lm.getLastKnownLocation(prov));
```

- Nella onResume() dell'Activity

```
lm.requestLocationUpdates(prov, 20000, 10, this);
```

- ... e nella onPause()

```
lm.removeUpdates(this);
```

- L'Activity dovrà implementare il listener

```
bla bla Bla extends Activity implements LocationListener {  
    public void onLocationChanged(Location l) {aggiorna(l);}  
    public void onProviderEnabled(String prov) { ; }  
    public void onProviderDisabled(String prov) { ; }  
    public void onStatusChanged(String prov, int status,  
        Bundle extras) { ; }  
    ...  
}
```

- A questo punto, la aggiorna() può implementare una delle strategie viste prima

```
void aggiorna(Location l) {  
    double lat = l.getLatitude();  
    double long = l.getLongitude();  
    float acc = l.getAccuracy();  
  
    /* decide se aggiornare current o meno */  
}
```

- Finalmente, il resto dell'app può usare current dove serve

PROXIMITY ALERT

Android consente anche di stabilire dei proximity alert. Il programmatore specifica latitudine e longitudine del punto centrale e il raggio dell'area "sotto osservazione". Definisce anche il PendingIntent contenente l'Intent da spedire tutte le volte che il dispositivo attraversa il confine dell'area sia in ingresso che in uscita. Il sistema seleziona automaticamente il provider "giusto" per tenere sotto osservazione l'area, se si è lontani dal bordo, usa un provider a basso costo e bassa precisione, man mano che ci si avvicina al bordo, aumenta la precisione e la frequenza dei controlli. Gli alert possono essere impostati con un timeout, se il device non attraversa il confine in

tempo non viene mandato l'alert. Per impostare un alert:

```
Intent i = new Intent(...);
PendingIntent pi = PendingIntent.getBroadcast(this, 0, i, 0);
lm.addProximityAlert(lat, long, raggio, timeout, pi);
```

L'intent spedito conterrà fra gli Extras una chiave KEY_PROXIMITY_ENTERING true se è nell'area, false altrimenti. Per cancellare un proximity alert invece lm.removeProximityAlert(pi);

Sui dispositivi che dispongono di Google Play Services è disponibile una libreria che semplifica l'ottenimento della posizione corrente. Ovviamente non tutti i dispositivi hanno Play Services poiché una libreria è non parte integrante del S.O. Essa offre tre tipi di servizi:

Fused Location Provider: implementa una strategia ottimale che combina caching, vari location provider e risparmio batteria.

Geofencing: è la versione ottimizzata dei Proximity Alert

Activity recognition: Classificazione delle attività dell'utente come camminata, corsa, bici ecc

GEOCODING

Il geocoding consiste nell'associare fra di loro due modi comuni di denotare posizioni geografiche cioè latitudine e longitudine o indirizzi postali. Gli indirizzi sono un sistema folle per il software quindi la traduzione viene effettuata su server aggiornati. Nel caso di Android i servizi di geocoding fanno parte delle API di Google Maps, sono quindi disponibili solo su device in cui oltre ad avere Android puro sono installati i Google Play Services. Nel nostro mercato quasi tutti i device ma in altri paesi come la cina è possibile che essi siano bloccati.

La classe Geocoder si occupa di tutto, anche dello scambio di dati con i server di Google, bisogna però che le applicazioni abbiano il permesso di accesso alla rete. La comunicazione con la rete è sincrona quindi non bisogna accedere al Geocoder dal thread UI. Il Geocoder dipende dal locale corrente (IT, US ecc..) e come detto prima non è un servizio di sistema ma si costruisce una istanza con al new. Una volta instanziato, per sapere se il geocoding è supportato sul particolare dispositivo si chiama Geocoder.isPresent() che restituisce una variabile booleana statica. Se il geocoding non è presente, sarebbe bello che l'applicazione facesse un graceful degrade cioè far utilizzare comunque l'app ma con funzioni limitate.

Forward Geocoding: da (indirizzo) stringa a lat-long

L'indirizzo è espresso come una stringa e si deve avvicinare al formato convenzionale il più possibile
getFromLocationName(indirizzostringa, maxreturnmatch);

Reverse Geocoding: da lat-long a stringa

Restituisce una serie di match con tutti gli indirizzi nelle vicinanze

```
getFromLocation(lat, long, maxreturnmatch);
```

CLASSE ADDRESS

Gli oggetti di classe Address rappresentano un indirizzo strutturato, esso contiene quante più informazioni possibili sulla denotazione di un luogo fisico. Il Geocoder cerca di compilare più campi possibili. I dettagli dell'indirizzo sono tutti rappresentati come stringhe, il metodo
getAddressLine(i) restituisce la i-esima riga nel formato usato solitamente sulle buste, metodi più specializzati restituiscono i campi in maniera "semantica" come getAdminArea(),
getCountryCode(), getCountryName()... e naturalmente, getLatitude(), getLongitude().

Esempio di geocoding asincrono:

```
private class ReverseGeocodingTask extends AsyncTask<Location, Void, Void> {
    Context mContext;
    public ReverseGeocodingTask(Context context) {
        super();
        mContext = context;
    }
    @Override
    protected Void doInBackground(Location... params) {
        Geocoder geocoder = new Geocoder(mContext, Locale.getDefault());
        Location loc = params[0];
        List<Address> addresses = null;
        try {
            addresses = geocoder.getFromLocation(loc.getLatitude(), loc.getLongitude(), 1);
        } catch (IOException e) {
            /* ... */
        }
        if (addresses != null && addresses.size() > 0) {
            Address address = addresses.get(0);
            String addressText = String.format("%s, %s, %s",
                address.getMaxAddressLineIndex() > 0 ? address.getAddressLine(0) : "",
                address.getLocality(),
                address.getCountryName());
            /* usa addressText, per esempio per aggiornare l'UI o come risultato */
        }
        return null;
    }
}
```

da developer.android.com

una volta definito il reversegeocodingtask possiamo invoca la risoluzione dell'indirizzo con

```
AsyncTask rgt = new ReverseGeocodingTask(this);
rgt.execute(new Location[] {location});
```

MAP VIEW

MapView è il widget che rappresenta una mappa, tuttavia l'interfaccia di questo particolare widget con il resto è complicata. L'interazione con il touch è complessa, con l'aggiunta del dialogo con i server di Google ed il caching. Nelle versioni di Android <3.0, viene fornita una Activity ad hoc chiamata MapActivity utilizzata se l'Activity contiene (solo) una MapView ed è ovviamente una sottoclasse di Activity. Essa gestisce tutto il ciclo di vita peculiare delle MapView, cura il dialogo con i server di Google tutto asincronamente. Varie API via web richiedono una chiave unica dello sviluppatore o dell'applicazione. In particolare per accedere alle API di Maps occorre registrarsi. La libreria che implementa MapView non fa parte di Android "standard", deve essere esplicitamente aggiunta al progetto e menzionata nel Manifest.

GEOPPOINT

È possibile controllare la visualizzazione di un MapView fornendo come parametri dei GeoPoint. Forma più precisa di coordinate geografiche formata da due interi espressi in microgradi (1 milionesimo di grado). Conversione da lat-long a GeoPoint:

```
Double lat = lat*1E6;
Double long = long*1E6;
GeoPoint gp = new GeoPoint(lat.intValue(), long.intValue());
```

La classe MapView offre diversi metodi per controllare cosa viene visualizzato, tutti passano per un MapController associato

```
MapView mv = findViewById(...);
MapController mc = mv.getController();
```

Fra i tanti metodi: mc.setCenter(gp) imposta il gp al centro, mc.animateTo(gp) idem ma con una animazione.

La MapView fa molto di più come permette di aggiungere overlay personalizzati alla mappa visualizzata. Sovrapponendo ad esempio percorsi o mappa meteo.

Il 3 Dicembre 2012, le “vecchie” API di GoogleMaps sono state deprecate, è stato possibile richiedere API key per le API v1 fino al 3 Marzo 2013 poi le “vecchie” applicazioni continueranno a funzionare e le “vecchie” chiavi saranno ancora valide ma ovviamente non ci sarà nessun aggiornamento futuro alle “vecchie” API. Le “nuove” API e key sono state rese disponibili dal Gennaio 2013, e sono ormai quelle “standard”. Le Maps API v2 Fanno parte dei Google Play Services e sono più flessibili poiché usano un MapFragment, hanno possibilità di 3D e le immagini sono vettoriali. La classe MapFragment rappresenta l’interfaccia utente standard per le mappe, La mappa in sé è invece rappresentata da un oggetto GoogleMap (associato al fragment).

Per ottenere un oggetto GoogleMap, si può chiamare il metodo getMap() del MapFragment. Il metodo può fallire e restituire null per molte ragioni, per esempio il Fragment non è ancora stato inizializzato oppure non sono installati i Google Play Services, oppure la vostra chiave sviluppatore non è valida ecc.. In questi casi, occorre in qualche modo avvisare l’utente e si può anche ritentare l’operazione più avanti. GoogleMap è un oggetto molto “delicato” quindi bisogna evitare di tenere riferimenti inutili, effettua numerose operazioni via rete quindi ha un comportamento asincrono sulla UI. Può essere acceduto solo dal thread UI quindi: runOnUiThread(), post(), postDelayed() e può essere necessario usare synchronized. In cambio, GoogleMap fornisce metodi per sovrapporre vostri contenuti alla mappa, gestire la visuale e l’aspetto, mappare fra coordinate geografiche e pixel su schermo, registrare un certo numero di listener (spostamenti camera inteso come parte visibile della mappa, click sulla mappa, click su marker ecc).

Riguardo il posizionamento della camera: nelle Maps v1, si impostava il centro mappa (lat,long), lo zoom, e il gioco era fatto. Nella Maps v2, si costruiscono istruzioni per lo spostamento della camera tramite la classe CameraUpdate, i CameraUpdate (prodotti da una classe factory) si applicano poi con:

animateCamera(): animazioni gestite autonomamente
moveCamera(): spostamento immediato

Esempio posizionamento camera:

```
MapFragment mfrag = ... ;
GoogleMap gmap = mfrag.getMap();
if (gmap!=null) {
    LatLng ll = new LatLng(lat,lng);
    gmap.animateCamera(CameraUpdateFactory.newLatLng(ll));
} else {
    /* aita! */
}
```

MARKER

E’ possibile anche aggiungere i propri marker ad una mappa. Si passa a addMarker() un oggetto MarkerOptions che specifica che marker vogliamo aggiungere. addMarker() restituisce un oggetto Marker che possiamo poi usare per riferire il marker creato, un tap sul marker avrà l’effetto di centrare la mappa sulla sua posizione. Un MarkerOptions specifica:

- Informazioni testuali come Titolo o snippet
- Informazioni grafiche come Icona
- Informazioni geografiche come coordinate geografiche espresse in Lat-Lon
- Comportamento come essere visibile o draggabile

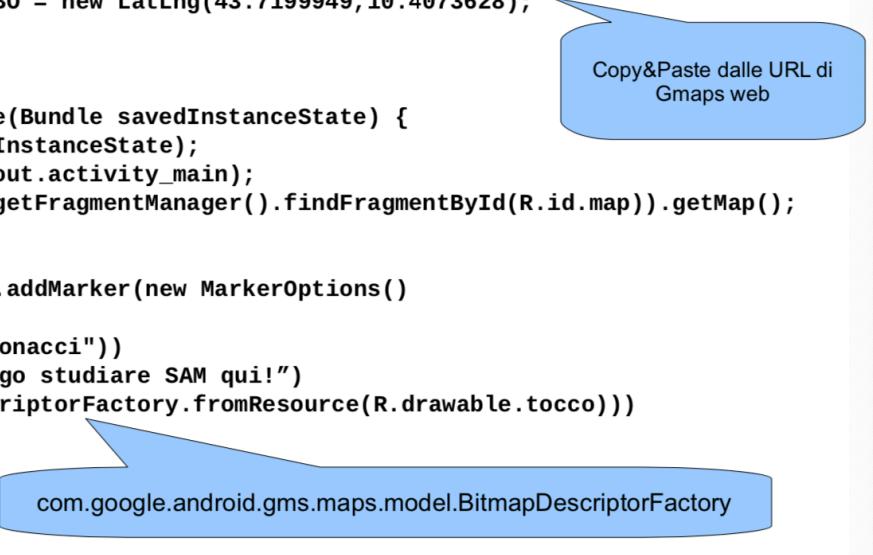
Un InfoWindowAdapter deve implementare due metodi: View getInfoWindow(Marker m) che dato un marker, restituisce una View per l'intero pop-up e View getInfoContents(Marker m) che dato un marker, restituisce una View per il contenuto del pop-up.

Esempio:

```
public class MainActivity extends Activity {
    static final LatLng FIBO = new LatLng(43.7199949, 10.4073628);
    private GoogleMap map;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        map = ((MapFragment) getFragmentManager().findFragmentById(R.id.map)).getMap();

        if (map!=null) {
            Marker mfibo = map.addMarker(new MarkerOptions()
                .position(FIBO)
                .title("Polo Fibonacci")
                .snippet("Che figo studiare SAM qui!")
                .icon(BitmapDescriptorFactory.fromResource(R.drawable.tocco)))
                .alpha(0.8));
        }
    }
}
```



ARCHITETTURA MULTIMEDIALE

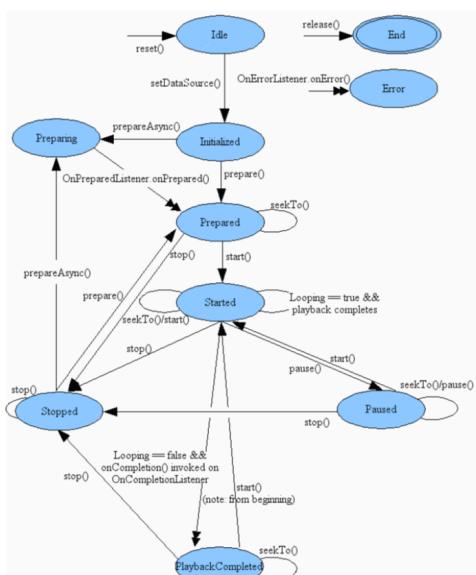
Android offre un framework sofisticato per il supporto ad audio, foto e video.

Classi principali (appartenenti ad android.media.*):

MediaPlayer: riproduce media

MediaRecorder: registra media

La riproduzione di media richiede di allocare e configurare un MediaPlayer descrivendo: da dove ottenere i dati da riprodurre e su quale dispositivo riprodurli. Eventualmente, controllando i dettagli: finestra da usare per il video, mixing dell'audio, ecc. Occorre poi controllare la riproduzione, gestita direttamente dall'utente: es: pulsanti Play/Pause/Stop oppure da programma: es: effetti sonori di un videogioco. E infine, liberare le risorse.



Il MediaPlayer è un automa con vari stati, inizialmente in Idle, attende la configurazione poi è Initialized. Segue il caricamento dei buffer o l'inizializzazione dello streaming (Preparing); quando tutto è pronto è Prepared. A questo punto, segue il ciclo classico del playback Started-Paused-Stopped. A fine media, va in Playback completed. Se si verificano errori, può passare nello stato Error, infine dopo il rilascio delle risorse va in End (stato terminale).

Il MediaPlayer incorpora decoder per molti formati comuni, in casi “strani”, si possono realizzare conversioni al volo da altri formati, formati “recenti” supportati solo su Android recenti. Le sorgenti dei dati possono essere ad esempio File Locali, URI locali e remote (URL), e anche risorse definite in R.raw.

Nel caso più semplice, è sufficiente creare un'istanza di MediaPlayer e avviare la riproduzione con i seguenti comandi:

```
MediaPlayer mp = MediaPlayer.create(this, R.raw.mymedia);
mp.start();
```

La risorsa dovrà essere un file in res/raw/mymedia.ext (secondo il formato), la chiamata a start() si occupa anche di preparare la riproduzione quindi non è necessario chiamare prepare().

Mentre la creazione è un'operazione istantanea, e l'avvio della riproduzione è un evento, la preparazione può richiedere parecchio tempo. Poiché nella prepare() si legge il file, si identifica il formato, nel caso di uri remote di apre una connessione TCP/IP ecc... Per questo motivo, la prepare() non deve mai essere fatta nel thread della UI ma va quindi chiamata su un altro thread. Direttamente, con Thread, Indirettamente, con AsyncTask e una volta tornati da prepare(), si notifica il thread principale. Il pattern comune, offerto da MediaPlayer è strutturato così: configuro la sorgente, registro un listener, chiamo prepareAsync() (che torna subito), quando il MediaPlayer è prepared, viene invocato un metodo del listener. La maggior parte dei metodi del MediaPlayer sono legali solo in certi stati come la prepare() solo nello stato Initialized. Ogni istanza di MediaPlayer occupa una gran quantità di memoria e blocca altre risorse come il Buffer audio/video e Socket per le connessioni di rete. È fondamentale rilasciarle dopo l'uso! Si consideri try/catch/finally contententi la mp.release().

Il caso della riproduzione video è simile a quello per l'audio, con le ovvie differenze. Occorre indicare una destinazione su schermo per il video, in uno di due modi:

```
setSurface(Surface s)
setDisplay(SurfaceHolder sh)
```

Il SurfaceHolder è tipicamente ottenuto da una SurfaceView inserita nel layout

```
sh = ((SurfaceView) findViewById(R.layout.video)).getHolder();
```

THE WAKE LOCK PROBLEM

Anche durante la riproduzione di audio o video, Android si sente libero di andare in modalità "sleep" (a basso consumo energetico) fermando tutto ma spesso questo non è desiderato! Ad esempio nella riproduzione di musica in background: si vuole spegnere lo schermo, ma continuare il play oppure nella riproduzione di video: si vuole mantenere acceso lo schermo e continuare il play. L'applicazione può chiedere un wake lock. Il wake lock è un concetto globale, ma il MediaPlayer fornisce metodi di utilità per ottenerlo tramite mp.setScreenOnWhilePlaying(b) mp.setWakeMode(ctx,flags) dove flags è tipo PARTIAL_WAKE_LOCK, FULL_WAKE_LOCK.

In alternativa, si può indicare che il dispositivo non deve mai andare in sleep se è visualizzata una certa Activity

Staticamente, nel file di layout

```
<RelativeLayout>
    android:keepScreenOn="true">
    ...
</RelativeLayout>
oppure dinamicamente, in Java
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

Ovviamente tenere lo schermo o la CPU sempre accesi senza un valido motivo, è una pessima idea. Assicuratevi di rilasciare il wake lock appena possibile e per gestioni più sofisticate potete usare il servizio di sistema PowerManager. Se la vostra applicazione manipola i wake lock, avrete

bisogno di chiedere il permesso all'utente, come al solito, in `AndroidManifest.xml`

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

MEDIACONTROLLER



Per controllare un MediaPlayer si può usare il widget di sistema MediaController viene associato a un'altra View e offre la UI per controllare la riproduzione.

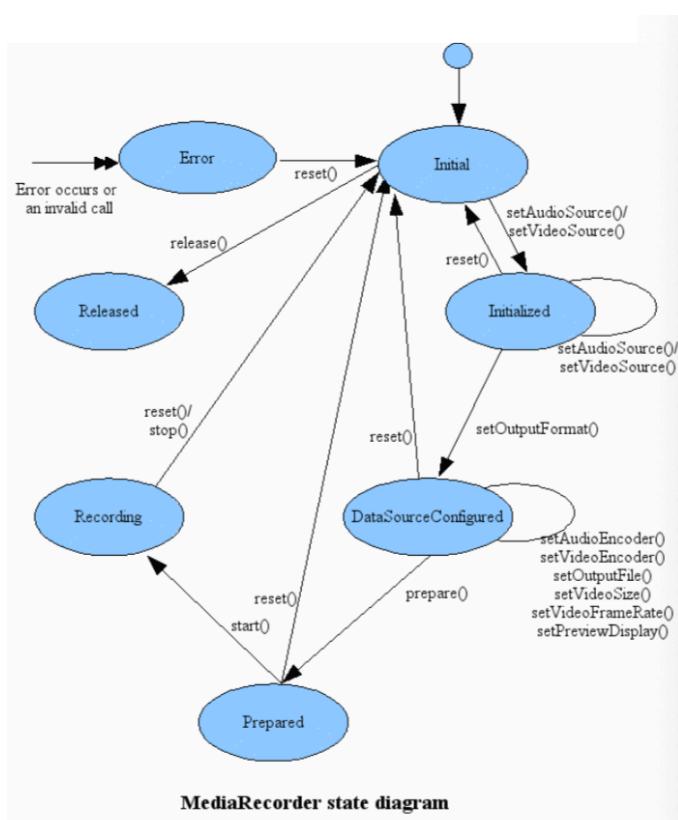
Tipicamente, si prepara un MediaPlayer mp e poi si implementa un MediaPlayerControl mpc dove si scrivono i metodi per le operazioni di start, pause ecc.. si inizializza un MediaController mc, lo si collega al MediaPlayer con `mc.setMediaPlayer(mpc)`, lo si collega a una vista con `setAnchorView(v)` dove di solito v è la SurfaceView connessa al mp. Una volta inizializzato, il MediaController si occupa di comparire in overlay su v quando l'utente tappa lo schermo (scompare dopo qualche secondo), di mantenere sincronizzato lo stato della UI con quello del mp e di passare i comandi dell'utente tramite UI al mpc che poi li passerà al mp.

SurfaceView connessa al mp. Una volta inizializzato, il MediaController si occupa di comparire in overlay su v quando l'utente tappa lo schermo (scompare dopo qualche secondo), di mantenere sincronizzato lo stato della UI con quello del mp e di passare i comandi dell'utente tramite UI al mpc che poi li passerà al mp.

VIDEOVIEW e STREAMING SU ALTRI DISPOSITIVI

Ancora più ad alto livello, è disponibile il widget VideoView. Incapsula praticamente tutta la gestione del video in una singola View. Però con limitate possibilità di configurazione. Android permette di fare streaming di contenuti su altri dispositivi (Chromecast) usando le classi MediaRouter e MediaRouterProvider.

MEDIARECORDER



L'analogo del MediaPlayer per la registrazione è una classe di nome MediaRecorder. Anche in questo caso, si crea un'istanza, la si configura, e la si usa per registrare. La configurazione è un po' più laboriosa, il MediaPlayer poteva estrarre parametri dai dati della sorgente (in base al formato audio/video). Per il MediaRecorder, i parametri vanno impostati dal programmatore che decide che tipo di file creare!

Anche il MediaRecorder ha una macchina a stati, dallo stato Initial, si passa a Initialized impostando una sorgente per la registrazione. Da lì, si passa a DataSourceConfigured impostando il formato di output, a questo punto, si chiama `prepare()` e si va in Prepared. Solo a questo punto si può iniziare la registrazione, passando in Recording. Al termine della registrazione, si torna in Initial.

La sorgente da cui registrare (audio o video) si imposta chiamando set AudioSource() e/o set Video Source(). In MediaRecorder.Video Source sono definiti solo due valori:

CAMERA: una fotocamera incorporata

DEFAULT: la camera di default

È però possibile scegliere una fotocamera, se ne è disponibile più di una, con set Camera(Camera c). La varietà di formati supportati dipende dalla versione di Android viene impostato con set Output Format() formati definiti da costanti in MediaRecorder.Output Format. Analogamente, è possibile impostare gli encoder audio e video.

Per l'audio: set Audio Encoder() mentre per il video: set Video Encoder().

L'output del MediaRecorder deve essere un file vero, questo impedisce di mandare i byte direttamente su buffer o Socket. Questo perché tutto il MediaRecorder è implementato in codice nativo (in C) per efficienza, e non in Java.

Il MediaRecorder deve essere creato in un thread che ha un Looper associato (in pratica: il thread della UI). Come al solito, passi come prepare() possono richiedere del tempo, e può essere importante eseguirli in modo asincrono.

È naturalmente possibile chiedere ad altre app di fare la registrazione di un video tramite intent e poi prendere il risultato nella on Activity Result.

```
Intent vi = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
if (vi.resolveActivity(getApplicationContext()) != null) {
    startActivityForResult(vi, 1);
}

@Override
protected void onActivityResult(int req, int res, Intent resi) {
    if (req == 1 && res == RESULT_OK) {
        Uri video = resi.getData();
        myVideoView.setVideoURI(video);
    }
}
```

ACCESSO ALLA FOTOCAMERA

Nella maggior parte dei casi, non è opportuno accedere direttamente alla fotocamera ma è meglio inviare un Intent dicendo che si vuole scattare una foto, e lasciare che parta l'app di fotocamera preferita dall'utente. Action predefinite: MediaStore.ACTION_IMAGE_CAPTURE e MediaStore.ACTION_VIDEO_CAPTURE. Si inviano con startActivityForResult() mentre nella onActivityResult() si ricevono i dati della foto scattata o del video ripreso dall'altra app.

L'alternativa consiste nello sviluppare una propria app di fotografia "incorporata" nell'app principale. Generalmente, non è una buona idea: troppo complicato ma se lo si vuole davvero fare, sarà necessario scoprire le caratteristiche hardware delle fotocamere disponibili sul dispositivo, collegare una camera a una SurfaceView per avere una preview, implementare una UI per impostare parametri fotografici e scattare le foto, ricevere i dati "grezzi" dalla fotocamera e/o convertirli in un formato umano e infine salvare i dati da qualche parte.

Con API <21 la classe principale da utilizzare è Camera. Fornisce metodi statici per enumerare le fotocamere disponibili. Si ottiene un'istanza con Camera.open(), poi si lavora con quella. Il metodo takePicture() "scatta" la foto, i dati vengono passati (come array di byte) al callback onPictureTaken() in maniera asincrona.

Con API >=21 una camera è vista come una pipeline di elementi, la classe CameraManager gestisce le fotocamere.

CameraManager cm = (CameraManager) getSystemService(CAMERA_SERVICE);

Ci sono vari metodi per ottenere l'elenco e le caratteristiche delle fotocamere come getCameraIdList() o getCameraCharacteristics(cids[0]);

L'app apre una connessione con la camera come visto precedentemente, scatta foto, i dati vengono passati al callback ecc...

Tutta l'interazione avviene tramite la callback, i cui metodi ricevono un CameraDevice:
onOpened(cd): camera aperta e pronta per l'uso
onClosed(cd): camera chiusa (in seguito a cd.close())
onError(cd, error): si è verificato un errore

NETWORKING TCP/IP

Il networking via TCP/IP su Android è standard, cioè si può usare quello che offre java. Il sistema operativo ottimizza il tutto per garantire la consegna dei pacchetti come il routing su reti di diverso tipo o la gestione dinamica al variare della connessione.

Server-side

```
try {
    ServerSocket ss = new ServerSocket(8080);
    while (!done) {
        Socket s = ss.accept();
        servi(s);
    }
} catch (...) { ... }
```

Client-side

```
try {
    Socket s = new Socket(server, 8080);
    ordina(s);
} catch (...) { ... }
```

Le letture e le scritture su rete avvengono tramite gli stream tipici di Java, dal socket si estraggono InputStream e OutputStream mentre letture e scritture avvengono ad esempio incapsulando gli stream in BufferedStream o OutputStream. Fondamentale il fatto che le operazioni su rete possono essere lente e/o bloccanti quindi mai accedere alla rete nel thread UI. E' ovviamente anche poco comune avere un server sul cellulare, perché per sua natura un server deve essere sempre in esecuzione (per ricevere notifiche push si utilizzano altre strade). Ancor più importante è il fatto che le operazioni su rete consumano molta energia (soprattutto in trasmissione) quindi è opportuno cercare di minimizzare il consumo trasferendo pochi dati o tenere una cache quando possibile, ancor di più fare coalescing (fare più comunicazioni tutte insieme).

CONNECTIVITYMANAGER

Il ruolo del ConnectivityManager è di fornire informazioni sulle reti accessibili al dispositivo. Si tratta dell'ennesimo servizio di sistema ottenibile con:

```
ConnectivityManager cm =
(ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);
```

Un oggetto NetworkInfo mantiene tutte le informazioni disponibili su un particolare network, ogni network quindi è rappresentato da un oggetto

```
NetworkInfo[] ni = cm.getAllNetworkInfo();
```

```
NetworkInfo currni = cm.getActiveNetworkInfo();
```

```
NetworkInfo wfni = cm.getNetworkInfo(NetworkInfo.TYPE_WIFI);
```

Oltre a TYPE_WIFI è possibile usare TYPE_BLUETOOTH, TYPE_ETHERNET, TYPE_MOBILE ecc...
Una volta ottenuto un NetworkInfo, si possono chiedere le caratteristiche di dettaglio della rete.
Informazioni del tipo getType(), getTypeName(), stato del tipo isAvailable(), isConnected(),
isFailOver(), oppure informazioni sugli errori come getReason().

Il ConnectionManager invia degli Intent broadcast (CONNECTIVITY_ACTION) ogni volta che cambiano le condizioni della rete. Si può registrare un BroadcastReceiver nel manifest, con un intent filter ma in questo caso, il vostro codice verrà eseguito sempre, anche quando l'app non è in esecuzione. Oppure, registrare dinamicamente il receiver nella onCreate() e deregistrarlo nella onDestroy(). Un'applicazione può ricevere notifiche che segnalano il passaggio a un nuovo tipo di rete per esempio il cambio da wi-fi a cellulare e viceversa. In molti casi, è utile modificare i pattern di accesso alla rete in base al tipo (e alle preferenze) come scaricare un podcast solo su wi-fi, ma ricevere le notifiche sulla disponibilità di una nuova puntata anche su 3G.

BLUETOOTH

Il Bluetooth è la più diffusa tecnologia per le personal area network, l'idea è di offrire connettività solo a dispositivi che sono in prossimità fisica alla persona dell'utente. Il protocollo è complicato e prevede molti profili per diverse classi di dispositivi. Ad esempio protocollo per auricolari diverso da quello per il trasferimento dei file. Ci interessa in particolare il profilo RFCOMM comunicazioni seriali in radiofrequenza. Bluetooth prevede che i dispositivi debbano essere accoppiati prima di poter scambiare dati a livello applicativo. Naturalmente, possono scambiare dati anche prima (per esempio, i loro nomi e classi), ma solo a livello di protocollo. L'accoppiamento deve confermare la volontà dell'utente/proprietario di entrambi i dispositivi, per questo motivo richiede in genere un segnale esplicito da parte sua: passkey (PIN). Un dispositivo Bluetooth può quindi essere in diversi stati:

Spento, modulo Bluetooth non attivato

Accesso ma non discoverable

Discoverable ma non paired

Discoverable e paired

Paired e non discoverable

In realtà il processo è più complesso: Frequency hopping su 12 canali diversi per area geografica, Point-to-point, piconet, scatternet.. Master/slave... ma fortunatamente se ne occupa il Sistema Operativo.

Nella maggior parte dei casi, si usano 4 classi

BluetoothAdapter: rappresenta la scheda di rete

BluetoothDevice: rappresenta un dispositivo

BluetoothServerSocket e BluetoothSocket: Analoghi a ServerSocket e Socket del networking TCP/IP

Esistono poi diverse altre classi specializzazioni per particolari profili Classi che descrivono i metadati dei profili ma non le vedremo... In teoria, un dispositivo potrebbe avere più adapter

Bluetooth, occorre quindi prendere una particolare istanza, per esempio:

BluetoothAdapter bta =BluetoothAdapter.getDefaultAdapter();

Se bta è null, il dispositivo non supporta BT, altrimenti si controlla se BT è abilitato, e in caso contrario si chiede all'utente di abilitarlo

```
if (!bta.isEnabled()) {  
    Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
    startActivityForResult(i, 1);  
}
```

L'utente a questo punto è libero di decidere se abilitare o meno BT, l'Activity che risponde all'intent è un Dialog di sistema. Verrà poi chiamata la onActivityResult() passando un risultato che indica l'esito:

```
void onActivityResult(int code, int res, Intent data) {  
    If (code==1) {  
        If (res==RESULT_OK) { /* BT abilitato! */ }  
        if (res==RESULT_CANCEL) { /* niet! */ }  
    } ...
```

In alternativa, è anche possibile registrarsi per ricevere l'Intent broadcast con action BluetoothAdapter.ACTION_STATE_CHANGED. Gli extra dell'Intent contengono informazioni sullo stato corrente. L'app può anche fare "piggybacking" cioè non richiede l'app di attivare il BT, ma è pronta a partire se qualcun altro lo attiva.

Il secondo passo consiste nello scoprire con quali BluetoothDevice possiamo comunicare. Con dispositivi paired in passato (e registrati):

```
Set<BluetoothDevice> devs = bta.getBondedDevices()
```

Con dispositivi discoverable e in range:

```
IntentFilter f = new IntentFilter(BluetoothDevice.ACTION_FOUND);
```

```
registerReceiver(this,f);
```

```
bta.startDiscovery();
```

Il nostro onReceive() riceverà tanti intent ACTION_FOUND quanti sono i dispositivi in range ciascuno ha negli extra una descrizione completa.

Esempio di receiver:

```
void onReceive(Context c, Intent i) {  
    String action = i.getAction();  
    if (BluetoothDevice.ACTION_FOUND.equals(action)) {  
        BluetoothDevice dev =  
            i.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);  
    } ...  
}
```

Per rendersi discoverable a propria volta:

```
Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);  
startActivityForResult(i, 2);
```

Una volta che è stato effettuato i pairing i due dispositivi possono comunicare, uno farà da server e l'altro da client. Il client è colui che inizia la connessione mentre il server è colui che sta ad aspettare una connessione. La connessione è ammessa solo se entrambi i device presentano lo stesso UUID (l'UUID può essere un accordo privato e svolge lo stesso ruolo del numero di porta in TCP/IP). L'intero processo è simile a quello per TCP/IP. Tuttavia, raramente il server rimane attivo in attesa di ulteriori connessioni più spesso si tratta di connessioni singole. Come al solito: mai nel thread UI! Il client deve specificare a quale device e servizio vuole connettersi, il device è uno dei device paired (ottenuto come visto prima) Il servizio è identificato dall'UUID.

Server:

```
String myname = "it.unipi.di.sam.btttest server";  
UUID myid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");  
BluetoothServerSocket bss =  
    bta.listenUsingRfcommWithServiceRecord(myname,myid);  
BluetoothSocket bs = bss.accept();  
bss.close();  
servi(bs);
```

Client:

```
String myname = "it.unipi.di.sam.btttest server";  
UUID myid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");  
BluetoothServerSocket bss =  
    bta.listenUsingRfcommWithServiceRecord(myname,myid);  
BluetoothSocket bs = bss.accept();  
bss.close();  
servi(bs);
```

Una volta che sia il server che il client hanno ottenuto un loro BluetoothSocket, il processo è il solito: Si ottengono InputStream e OutputStream per ogni socket, direttamente o tramite wrapping si procede allo scambio di dati, problemi di rete si traducono in fallimenti delle letture/scritture o eccezioni. Al termine, si chiama close() per chiudere la RFCOMM.

WIFI-DIRECT

Wi-fi direct è una tecnologia relativamente nuova per le connessioni point-to-point via wi-fi (è disponibile da Android 4.0 in poi). I casi d'uso sono analoghi a Bluetooth, ma molto più veloce e con range più esteso. I dispositivi devono avere entrambi wi-fi ma consente anche di creare gruppi p2p cioè dispositivi multipli, tutti in range, che comunicano liberamente. A differenza dei network layer precedenti, lo strato wi-fi direct di Android è stato sviluppato nativamente. Fa quindi uso del design più tipico di Android: si lanciano intent per chiedere azioni, si ricevono notifiche via listener. La classe WifiP2pManager fornisce i metodi che semplificano la vita. In effetti, il nome originale della tecnologia era "Wi-Fi P2P", quando il P2P era cool. Quando sono cominciate le cause contro i pirati del P2P, il nome "WiFi Direct" improvvisamente suonava meglio...

Il solito servizio di sistema ottenibile con

```
WifiP2pManager wfmd = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
```

L'inizializzazione del sistema ci fornisce un canale da usare poi come handle

```
Channel ch = wfmd.initialize(); I metodi del WifiP2pManager consentono di chiedere le principali funzioni, le risposte arriveranno tramite intent broadcast o attraverso chiamate a dei listener.
```

- Metodi principali:

Method	Description
initialize()	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi Direct method.
connect()	Starts a peer-to-peer connection with a device with the specified configuration.
cancelConnect()	Cancels any ongoing peer-to-peer group negotiation.
requestConnectInfo()	Requests a device's connection information.
createGroup()	Creates a peer-to-peer group with the current device as the group owner.
removeGroup()	Removes the current peer-to-peer group.
requestGroupInfo()	Requests peer-to-peer group information.
discoverPeers()	Initiates peer discovery
requestPeers()	Requests the current list of discovered peers.

Il processo di Discovery inizia con una chiamata a discoverPeers():

```
wfmd.discoverPeers(ch, new WifiP2pManager.ActionListener()
{
    @Override
    public void onSuccess() {
        ...
    }
    @Override
    public void onFailure(int reasonCode) {
        ...
    }
});
```

Una volta che il processo di discovery è concluso, il sistema invia un broadcast intent WIFI_P2P_PEERS_CHANGED_ACTION. A questo punto, la nostra applicazione (che avrà "visto" l'intent tramite un receiver) può chiamare requestPeers().

```
void onReceive(Context c, Intent i) {
    String action = i.getAction();
    if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
        wfmd.requestPeers(ch, pllist);
    }
}
```

Quest'ultimo listener sarà un WifiP2pManager.PeerListListener con unico metodo: onPeersAvailable(WifiP2pDeviceList peers). Ora peers contiene la lista dei dispositivi raggiungibili ed è possibile iniziare una connessione. Assumendo che dev sia un dispositivo dalla lista, possiamo creare una configurazione corrispondente, ed effettuare una connessione:

```
WifiP2pConfig cfg = new WifiP2pConfig();
cfg.deviceAddress = dev.deviceAddress;
wfdm.connect(ch, cfg, new ActionListener() {
    @Override
    public void onSuccess() {
        ...
    }
    @Override
    public void onFailure(int reason) {
        ...
    }
});
```

Una volta che i dispositivi sono connessi tra di loro si trasferiscono i dati come normali socket TCP/IP.

NFC

Le comunicazioni NFC (near field communication) sono comunicazione che si attuano ad una distanza massima: 4cm (a contatto).

Tre casi d'uso:

Lettura / scrittura di tag NFC, NFC P2P e Simulazione di un tag NFC.

I tag NFC fungono da “memoria di massa” con capacità molto limitata, le applicazioni possono registrarsi per essere attivate quando il dispositivo è posto a contatto con un tag NFC.

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

L'intent contiene fra gli extra le informazioni del tag.

PROGRAMMARE ANDROID IN CODICE NATIVO

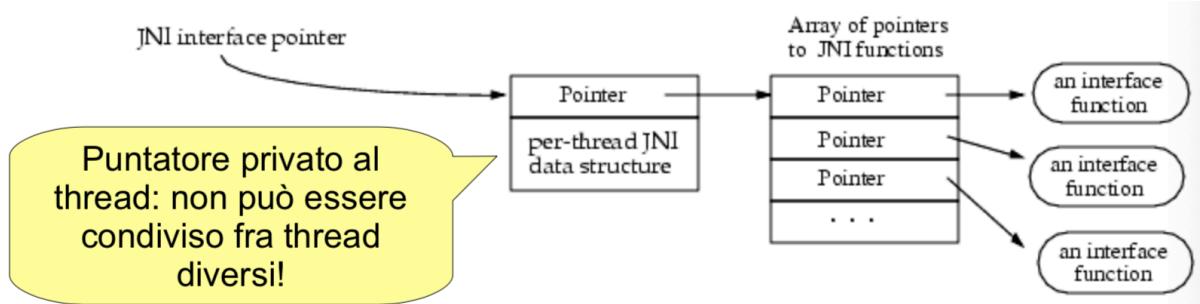
Android incoraggia l'uso di codice Java (compilato in DEX) per favorire la portabilità e la sicurezza vista l'esecuzione all'interno di una VM. Tuttavia, in certi casi la programmazione nativa è indispensabile ad esempio per ottenere prestazioni più alte, per utilizzare librerie pre-esistenti scritte in C / C++ o per l'accesso a livello di bit all'hardware. Gli svantaggi però sono svariati, in primis la complessità di sviluppo, assenza di portabilità (è necessario fornire versioni compilate del codice nativo per ciascuno dei processori su cui la mia app girerà). Android supporta la scrittura di applicazioni in C tuttavia è normalmente poco praticabile, usata solo in alcuni framework per giochi o applicazioni di controllo automatico. Più spesso si realizzano solo librerie in C che si occupano di far eseguire il codice “delicato”. Per produrre codice nativo per Android, è necessario installare una toolchain aggiuntiva rispetto al solito: NDK – Native Development Kit.

Java prevede già la possibilità di interfacciarsi con codice nativo, tipicamente, ma non necessariamente, scritto in C. In sostanza: moduli oggetto (file.o) contenenti codice compilato, e con le convenzioni di chiamata del C. Parametri passati tramite lo stack, Push e pop a cura del chiamante. Codice nativo raccolto in una libreria (file.so). La Java Native Interface (JNI) è l'insieme di specifiche che descrivono come interfacciare codice Java e C. È una parte standard del linguaggio Java ed è usata per implementare tutta l'interfaccia con il sistema operativo ospite. (Accesso ai file, alle socket, alle primitive grafiche...)

JNI fornisce funzioni C per:

- Creare oggetti Java, leggere e scrivere valori nei loro campi
- Chiamare metodi statici e di istanza
- Generare e catturare eccezioni
- Caricare classi Java a runtime e ispezionarle tramite reflection
- Controllare a runtime i tipi “veri” degli oggetti

Le funzioni offerte da JNI (da chiamare in C) sono offerte tramite un puntatore a un array di puntatori a funzione, il puntatore da usare viene passato come argomento al vostro codice C.



Le funzioni C scritte dal programmatore devono essere compilate in una libreria a caricamento dinamico *.so, visto che Android gira su kernel Linux devono essere rientranti e thread-aware ovvero, eseguibili contemporaneamente da più thread (opzione del compilatore -D_POSIX_C_SOURCE). Il file *.so deve essere incluso nel vostro progetto nella directory lib/. È necessario che la vostra libreria C sia caricata in memoria prima di poterne chiamare le funzioni. Java offre il metodo statico System.loadLibrary() per essere sicuri che la libreria sia caricata al momento giusto. Solitamente si inserisce la loadLibrary() in un inizializzatore statico della classe, cosicché quando la VM carica la vostra classe, esegue anche il loadLibrary().

In Java, il qualificatore native indica al compilatore che un metodo non ha un corpo proprio. All'invocazione del metodo, verrà eseguita una funzione C associata, il cui codice si trova in una libreria caricata in precedenza. La corrispondenza dei nomi fra metodo Java e funzione C è data da un processo di mangling di cui esistono due forme alternative:

1. Se un metodo è overloaded, il mangling incorporerà la sequenza di descrittori di tipo degli argomenti (nome lungo).
2. Altrimenti, viene effettuato solo il mangling del nome del metodo (nome breve).

La VM prova prima a cercare il nome “breve”, se non lo trova cerca quello “lungo”.

Visto che il nome è a nostra discrezione... Keep It Super Simple.

C'è un piccolo problema perché Java consente di avere pressoché tutti i caratteri di UNICODE in un identificatore, mentre C consente solo ASCII 7 bit. Una parte del processo di mangling viene quindi applicata a identificatori come Nomi di package, di classi, di metodi. Quindi si procede così:

I tipi T sono indicati da un codice alfabetico

- Z – boolean
- B – byte
- C – char
- D – double
- F – float
- I – int
- J – long
- S - short
- V – void
- L – object
 - seguito dal nome fully qualified di una classe
 - elementi separati da / e terminato da ;
- [– array
 - Seguito dalla codifica del tipo degli elementi

```
package it.unipi.di.sam.nativa;  
class Nativa {  
    native int somma(int a, int b);  
}
```

Java_it_unipi_di_sam_nativa_Natива_somma_II

Alla funzione C (quella con il nome mangled) vengono passati due argomenti in più rispetto a quelli del metodo Java: un puntatore alla struttura JNIEnv e un puntatore a this. Tutti gli altri argomenti sono marshalled cioè convertiti da valori Java a valori C. Spesso, con limitazioni...

L'header jni.h definisce nomi C per i tipi Java

Tipo Java	Tipo C	Note
boolean	jboolean	Unsigned, 8 bit
byte	jbyte	Signed, 8 bit
char	jchar	Unsigned, 16 bit
short	jshort	Signed, 16 bit
int	jint	Signed, 32 bit
long	jlong	Signed, 64 bit
float	jfloat	32 bit
double	jdouble	64 bit
void	jvoid	Non esistono valori

In definitiva:

```
package it.unipi.di.sam.nativa;
class Nativa {
    native int somma(int a, int b);
}

#include <jni.h>
jint Java_it_unipi_di_sam_nativa_Nativa_somma_II(
    JNIEnv *env, jobject this, jint a, jint b) { ... }
```

ACCEDERE A JAVA DA C

Se la vostra funzione si limita a leggere i valori degli argomenti (di tipi base), fare qualche conto, e restituire un risultato, la situazione è semplice. Il 90% degli esempi che si trova in giro è così. char, int, double ecc. vengono copiati fra Java e C. Tutti questi accessi devono essere mediati da apposite funzioni offerte da JNIEnv. Ma in casi reali, dovete fare accesso agli oggetti Java da C. Per accedere ai campi e ai metodi di un oggetto Java (di cui si ha il riferimento) occorre usare degli identificatori numerici (in realtà, sono puntatori C)

jfieldID – id di un campo

jmethodID – id di un metodo

I valori vengono ottenuti chiamando funzioni di JNIEnv.

jfieldID GetFieldID(JNIEnv *env, jclass clazz, const char *name, const char *sig);

Una volta ottenuto il jfieldID di un campo, si legge il valore con un'altra funzione di JNIEnv

NativeType Get<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID);

Esempio

```
class FieldAccess {  
    static int si;  
    String s;  
  
    private native void accessFields();  
  
    public static void main(String args[]) {  
        FieldAccess c = new FieldAccess();  
        FieldAccess.si = 100;  
        c.s = "abc";  
        c.accessFields();  
        System.out.println("J FieldAccess.si = " + FieldAccess.si);  
        System.out.println("J c.s = \"" + c.s + "\"");  
    }  
  
    static {  
        System.loadLibrary("MyImpOfFieldAccess");  
    }  
}  
  
#include <stdio.h>  
#include <jni.h>  
#include "FieldAccess.h"
```

Parte Java

```
JNIEXPORT void JNICALL Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)  
{  
    jclass cls = (*env)->GetObjectClass(env, obj);  
    jfieldID fid;  
    jstring jstr;  
    const char *str;  
    jint si;  
  
    fid = (*env)->GetStaticFieldID(env, cls, "si", "I");  
    if (fid == 0) return;  
    si = (*env)->GetStaticIntField(env, cls, fid);  
    printf("C FieldAccess.si = %d\n", si);  
    (*env)->SetStaticIntField(env, cls, fid, 200);  
  
    fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");  
    if (fid == 0) return;  
    jstr = (*env)->GetObjectField(env, obj, fid);  
    str = (*env)->GetStringUTFChars(env, jstr, 0);  
    printf("C c.s = \"%s\"\n", str);  
    (*env)->ReleaseStringUTFChars(env, jstr, str);  
    jstr = (*env)->NewStringUTF(env, "123");  
    (*env)->SetObjectField(env, obj, fid, jstr);  
}
```

Parte C

La funzione di JNIEnv GetJavaVM restituisce un puntatore a una struttura JavaVM che “rappresenta” la macchina virtuale Java. JavaVM offre a sua volta una tabella di puntatori a funzione che consentono di manipolare la VM. Tuttavia, Android non usa la JavaVM! Non sempre la specifica JNI e il comportamento di Android sono consistenti. In particolare: Android usa una sola VM per tutto, con il metodo Dalvik, o precompila con ART. La maggior parte dei valori usati nel codice C saranno riferimenti locali validi per tutto il tempo di esecuzione della funzione C. Se però allocate nuovi oggetti, e volete restituirli al chiamante, non potranno essere riferimenti locali verrebbero distrutti all'uscita. Per questo motivo si possono dichiarare riferimenti globali che però vanno deallocati a mano: niente garbage collection.

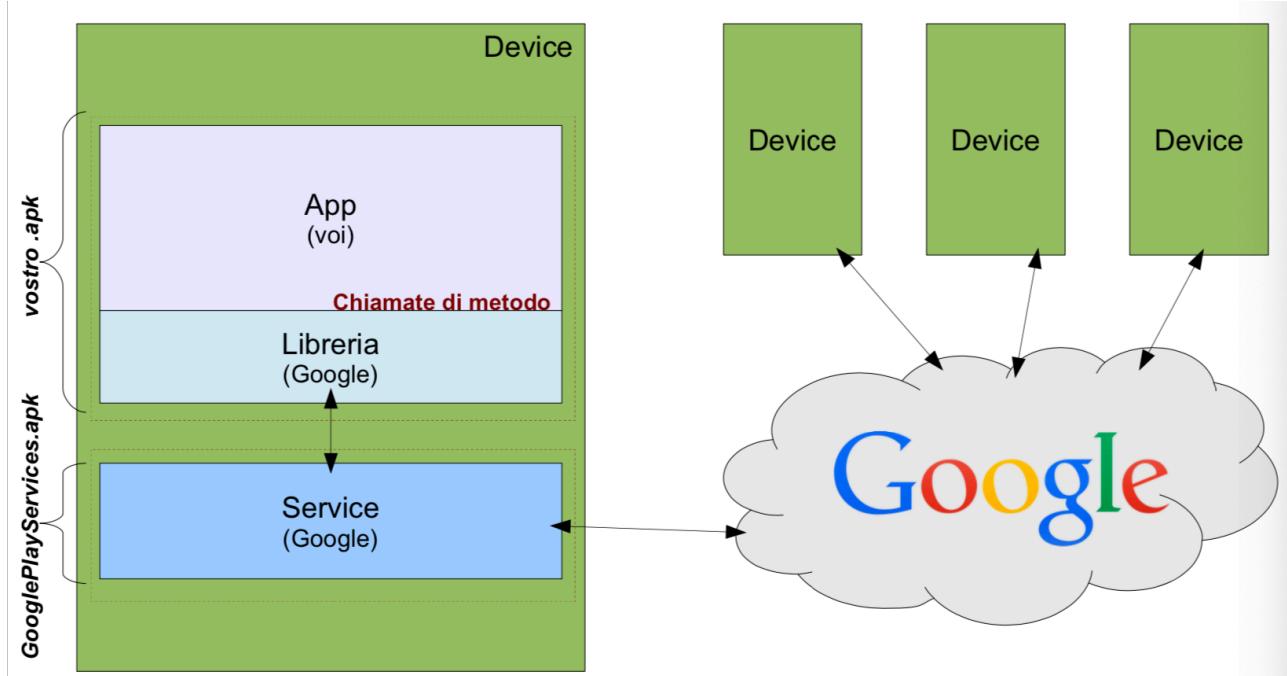
Consiglio: cercare di strutturare l'interfaccia fra Java e C in modo che tutta la parte “complicata” sia fatta in Java. La parte in C dovrebbe solo fare number crunching su tipi base, oppure accesso all'hardware. Non appena avete oggetti o eccezioni che attraversano l'interfaccia Java/C, correte il rischio di fare grossa confusione soprattutto il codice C è unsafe! Potete andare in crash!

GOOGLE APIs

Oltre 100 APIs ciascuna con dozzine o centinaia di metodi. Così tanti servizi da richiedere un motore di ricerca interno con una sezione delle API più popolari (API EXPLORER). Ogni API offre un insieme di metodi che possono essere chiamati in stile REST. Ogni chiamata include una richiesta (HttpRequest), tipicamente con un payload JSON che fornisce gli argomenti. La risposta è una

coppia <codice,corpo> in cui il codice è un error code HTTP (200=ok, 404=forbidden, ecc.), mentre il corpo è un oggetto JSON i cui campi rappresentano il risultato della chiamata.

Architettura delle APIs Google



La gestione degli errori invece è complicata, dopo una chiamata a `gapi.connect()` può darsi che sia andato tutto a buon fine, oppure qualcosa sia andato storto (`onConnectionFailed(res)`). In caso di fallimenti, è possibile che il sistema offra una risoluzione implicita come ad esempio il client non era autenticato. Però spesso le risoluzioni necessitano di azione dell'utente.

Le operazioni su GoogleAPI sono per loro natura asincrone e fallibili poiché passano da rete, sotto c'è REST. Tutte le chiamate vengono fatte tramite metodi statici di classi di libreria corrispondenti alle API. In molti casi, restituiscono un `PendingResult` su cui si può agire in tre modi:

registrando una callback da chiamare quando il risultato è pronto usando `setResultCallback` sul `PendingResult`. Oppure sospendendo il thread in attesa che il risultato sia pronto chiamando la `await()` (ovviamente non nel thread UI). Oppure cancellando l'operazione (e testando se è stata cancellata) usando la `cancel()`.