Sviluppo Applicazioni Mobili

Appunti Riassuntivi

Raffaele Apetino, Matricola 549220

STORIA DI ANDROID

Si cercò di creare un sistema operativo per telefoni che non fosse fortemente proprietario come il blackberry o windows CE. Nel 2007 si forma la Open Handset Alliance, e 7 giorni dopo viene rilasciato il primo SDK (Software Development Kit) basato su linux 2.6. 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. I produttori hanno poco interesse nell’aggiornare i telefoni vecchi, facendone comprare di nuovi agli utenti, abbiamo così una frammentazione che rende difficile lo sviluppo e la retrocompatibilità. Ovviamente sono presenti delle API, identificate da un API Level. (level 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 in Java in ambiente Android. Ci sono 3 ambienti di sviluppo per Android:

1) CLI (linea di comando - toolchain)

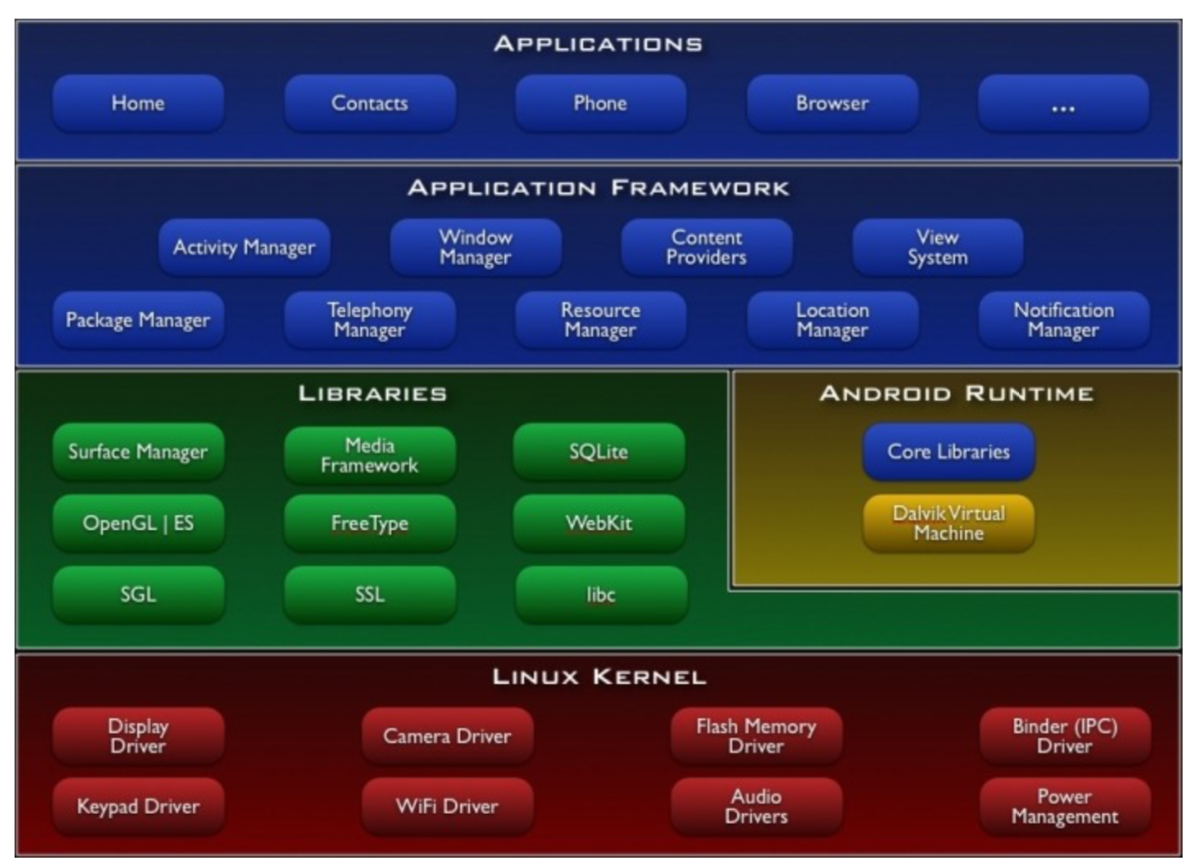
2)Eclipse con plug-in

3)Android Studio (versione customizzata di IntelliJ).

Su Eclipse si usa un AndroidDevelopmentTool Bundle che contiene tutto il necessario.

Android Studio invece è disponibile come solo bundle. Android studio usa Gradle: un sistema di build 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. L’emulatore è una macchina virtuale configurabile, se l’emulatore è configurato su una immagine ARM, viene eseguita in modo interpretato (lento), se l’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 (utilizzata dalla creazione di android fino alla versione 4.4), con nome Dalvik. E’ una versione della JVM basata su registri. 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 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, questo controllo è fatto dal kernel.

I veri utenti delle applicazioni sono i programmatori delle app, il proprietario dello smartphone è quasi un device, 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.

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.

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 e gli assets (altri dati). Gli assets sono aggiunti semplicemente all’APK e acceduti come file normali. Durante il processo di build il tool “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 degli 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 i file stessi XML /res/menu/laymenu.xml)

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

In XML @[package]timp/nome mentre in 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 modificge: 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, 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 altrew activity, recuperare o salvare dati tramite content provider, terminare activity ecc…

AVVIO DI UNA ACTIVITY

Explicit intent: creiamo un intent che chiede una activity

Implicit intent: creiamo un intent che chiede una funzione

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 l’azione che vuole eseguire, Data su quali dati operare, component per il componente a cui è indirizzato il messaggio, ecc…

Ogni app quindi definisce un Intent Filter in AndroidManifest che indica quali messaggi vogliamo ricevere.

ANDROID MANIFEST

Contiene:

la 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 stitica e venivano chiesti a tempo di installazione)

elenco dei componenti dell’applicazione

altri metadati: librerie necessarie…

ACTIVITY

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

Il comportamento dell’activity in fase di avvio/riavvio può essere alterato dal programmatore tramite i vari FLAG\_ACTIVITY\_\* dell’intent che lancia una activity.

Es: 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

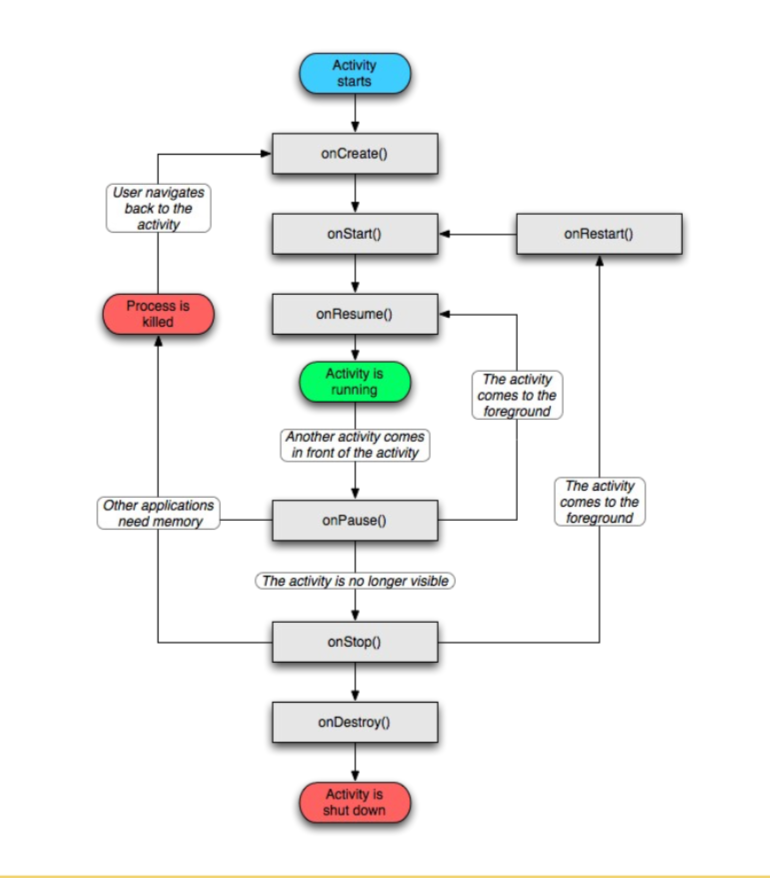
Nel manifest è presente la pima activity di un App, non esiste un main ma piuttosto tutte le activity che accettano un intent MAIN/LAUNCHER sono potenziali Main. Alcune Activity hanno senso solo se possono inviare un risultato di qualche tipo a chi le ha invocate (startActivityForRestult()). Per esempio: 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 incapsulano la risposta. Normalmente una activity ha un layout che si setta con setLayout(), nel secondo caso piuò essere anche formata da un fragment 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 autoterminarsi 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’activiry 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 disallocata 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 dei metodi per far si 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. C’è una corrispondenza tra XML e metodi in java per modificare le nostre view o layout.

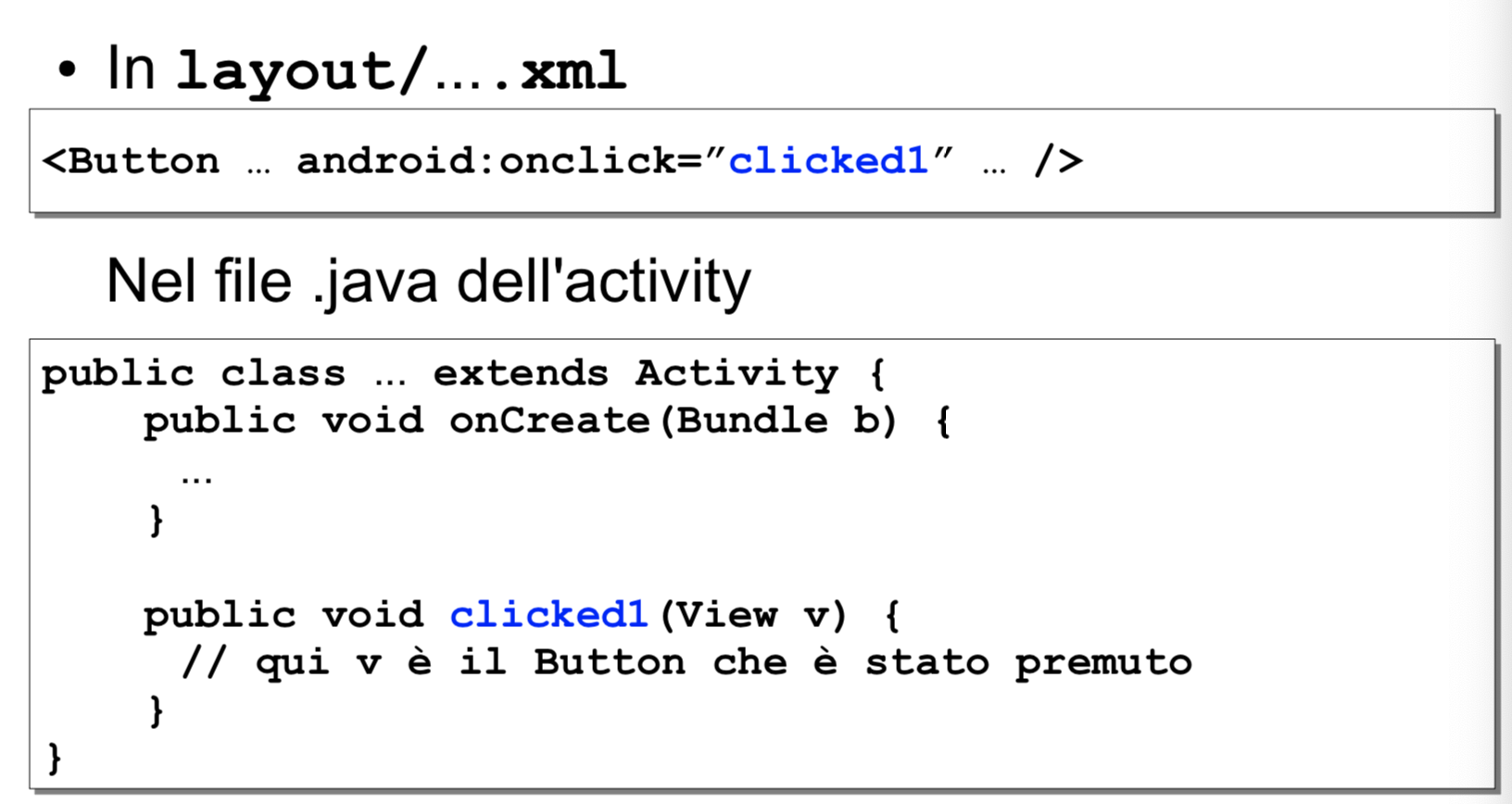
GESTIONE DELL’INPUT

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

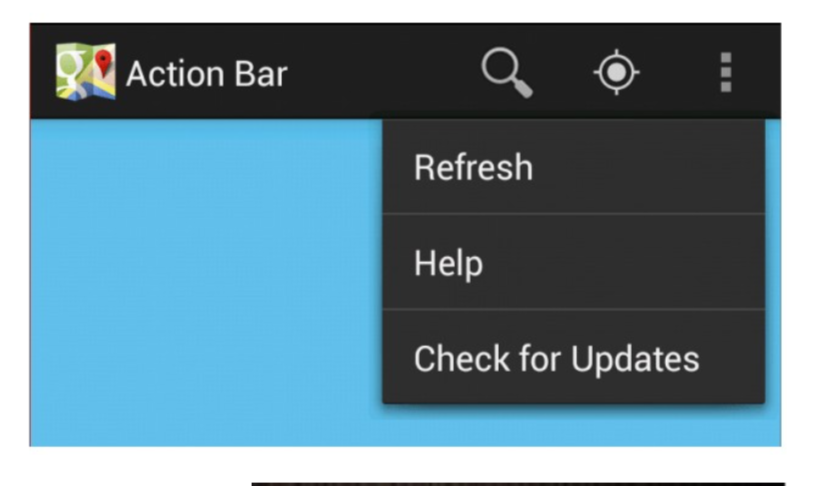
onDragListener(), onGenericMotionListener(), onKeyListener(), onLongClickListener() ecc… è possibile anche definire nell’XML il nome del metodo da chiamare in risposta ad un evento:



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 opzionalmente 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 sottomenù (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/ contente 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

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 che abbiamo visto finora dipendevano dall'Activity, essendo più “globali”.

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 onCreateContextMenu() 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:

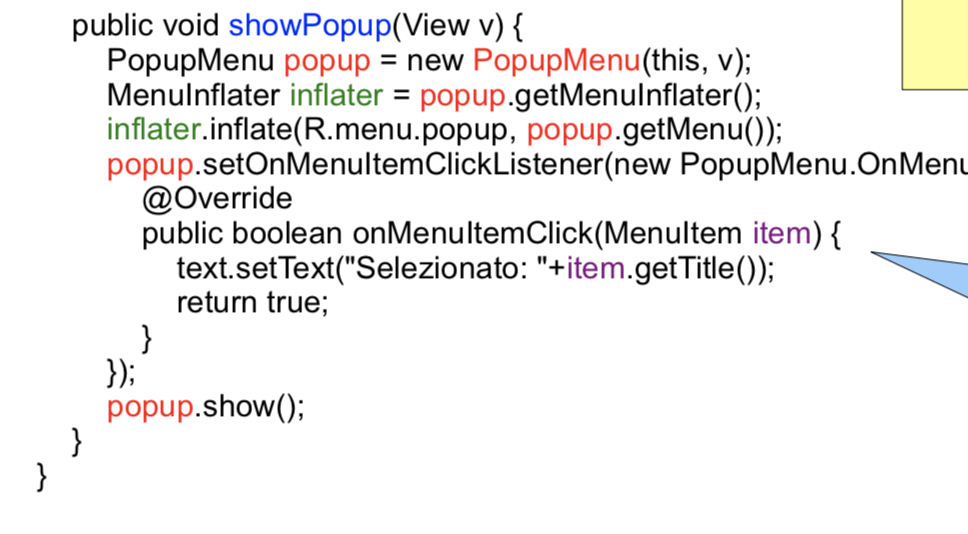
* Registrare un intent direttamente nel MenuItem
* Registrare un MenuItemClickListener nel MenuItem
* Implementare onContextItemSelected(MenuItem menuitem) nell'Activity, come al solito, si discrimina poi in base all'ID di menuitem. (Quest'ultimo è il metodo più efficiente)

Come alternativa ai context menu, è possibile utilizzare una action bar contestuale, 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, Il pannello compare subito sotto la View.

(Se c'è spazio: altrimenti, prova sopra).



CREAZIONE MENU DINAMICA IN BASE A INTENT

È possibile chiedere al sistema di riempire un nostro menu con tutte le azioni offerte da componenti del sistema sui dati che noi specifichiamo. 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 sono i requisiti la onMeasure deve fornire la risposta chiamando il metodo setMeasuredDimension, tipicamente si chiama la super 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 (superficie di disegno. Il Canvas 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

Non esiste View() perché non si può avere una view senza il suo contesto.

SCRIVERE UN PROPRIO LAYOUT

Si può ereditare da viewGroup e si crea un gruppo di widget preassemblato (public class EqLayout extends ViewGroup) , oppure si può creare un layout custom contentendo di disporre i figli view secondo criteri personalizzati. Da usare quando i LayoutManager disponibili non soddisfano le necessità.

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 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, dovrà anche fare override di onMeasure() perché le dimensioni del gruppo dipenderanno dalle dimensioni dei figli e dalla disposizione degli stessi.

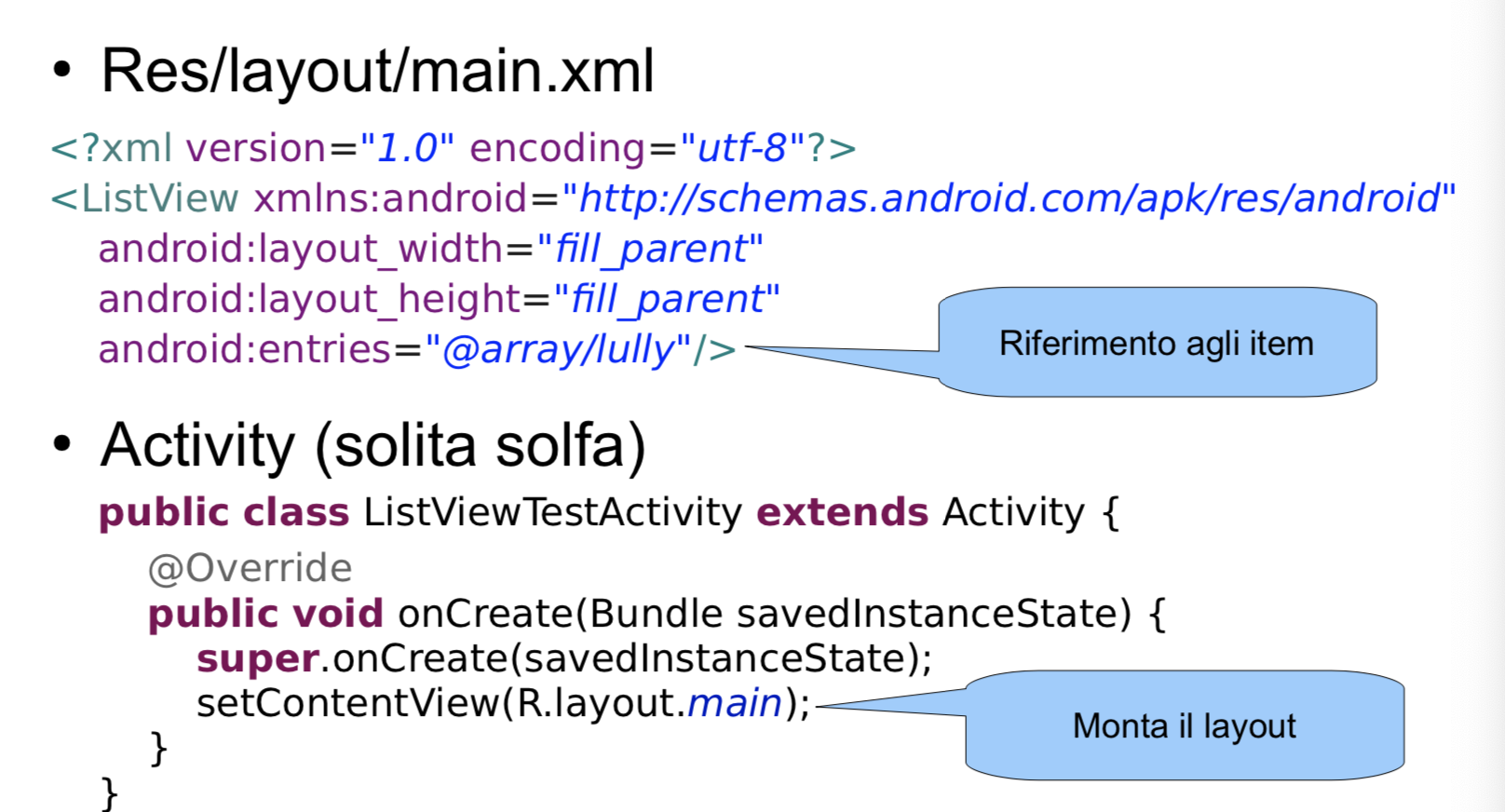
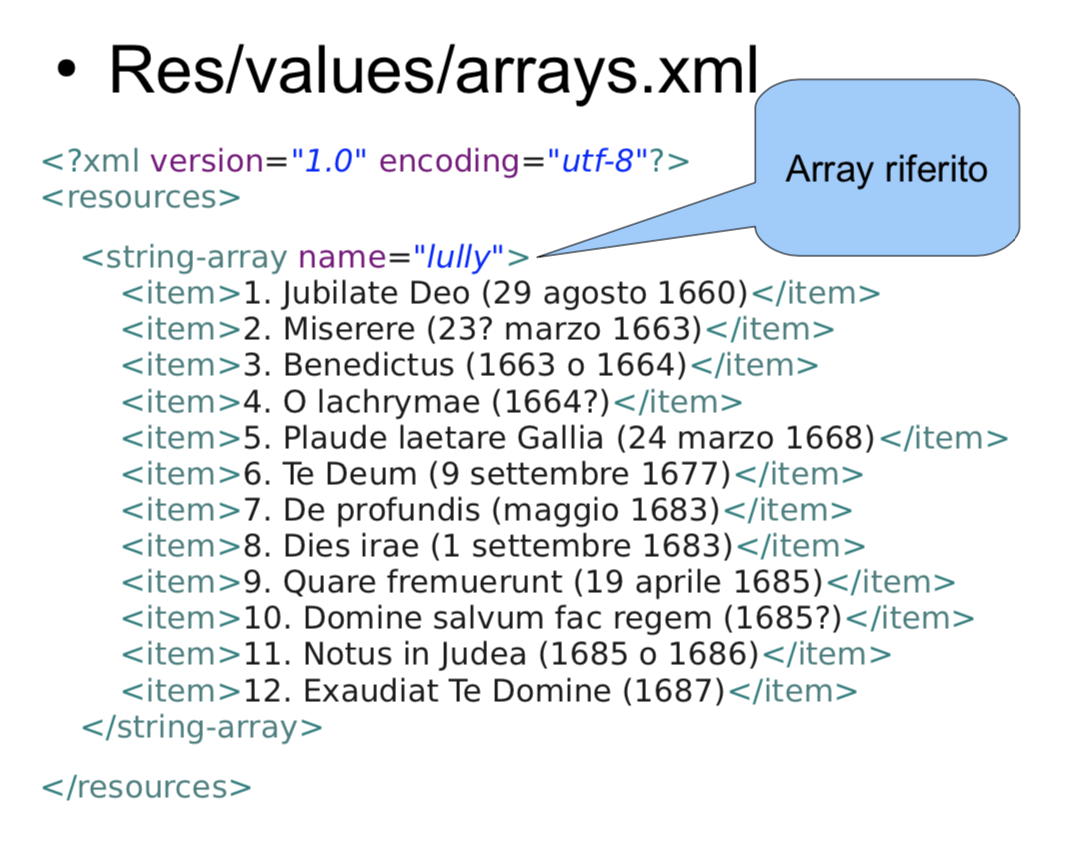
STILI E TEMI

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, riferendo lo stile definito. Analogamente, per applicare un tema si usa l'attributo android:theme in AndroidManifest.

LISTVEW E DATA ADAPTER

Uno dei componenti più utilizzati in una interfaccia grafica su android è una 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.



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, fornire la view alla listview cui adapter è associato e notificare gli 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.

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 sottoclasse di Activity specializzata per contenere ListView. Il layout di default contiene due view:

La ListView, con id “@android:id/list”

Opzionalmente, una view per il caso di lista vuota, con id “@android:id/empty”.

È anche possibile usate 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 ricilando le View per evitare new e risparmiare memoria.

Però questo comportamento è strettamente accoppiato con la gestione a lista, visualizzazione con LinearLayout verticale, Scroll up & down. Recycleview separa queste due funzioni. RecyclerView fornisce:

Un LayoutManager per decidere come disporre gli elementi (linear, 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

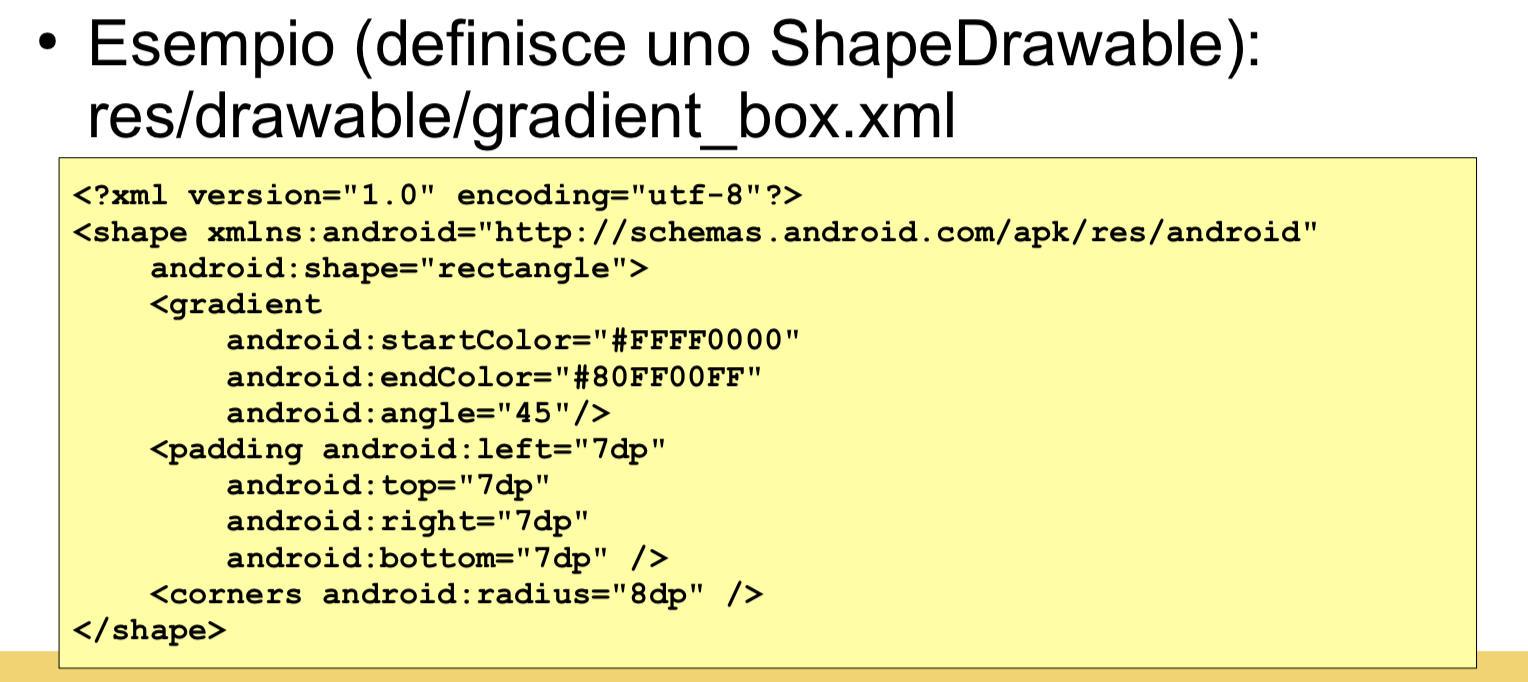
Una webview è una view che incapsula un web browser. Fornisce tutte le funzioni base per caricare una URL e visualizzarla all’utente. Fornisce anche funzioni più specifiche per intercettare funzioni particolari: esecuzione Javascript, gestione cookie… E’ possibile anche eseguire il browser esternamente 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 come un normale accesso alle risorse e usato per disegnare su un canvas.



Una immagine .jpg, .png 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 chiamanto Asset Studio.

NOTIFICHE ALL’UTENTE.

Android prevede tre forme di notifica all’utente:

Toast e SnackBar: brevi Pop-up che appaiono sopra l’activity e scompaiono aitomaticamente dopo poco tempo

Toast: semplici e veloci, poco intrusivi 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();

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 prograss bar va animata a mano, post lollipop si usa setProgress)

Dialog: finestre pop-up tradizionali con possibilità di intertazione dell’utente.

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 scorrerà̀ 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)

Però le notifiche ora hanno molti più̀ metadati che le collegano al contesto mentale dell'utente Persona (contatto) di riferimento

App che l'ha prodotta

Classificazione delle informazioni

Sensibili: da non mostrare sul lock screen

Compatte: da mostrare nel notification drawer con la notifica collassata

Estese: da mostrare nel notification drawer con la notifica espansa

In più: informazioni sulla propagazione su altri device

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:

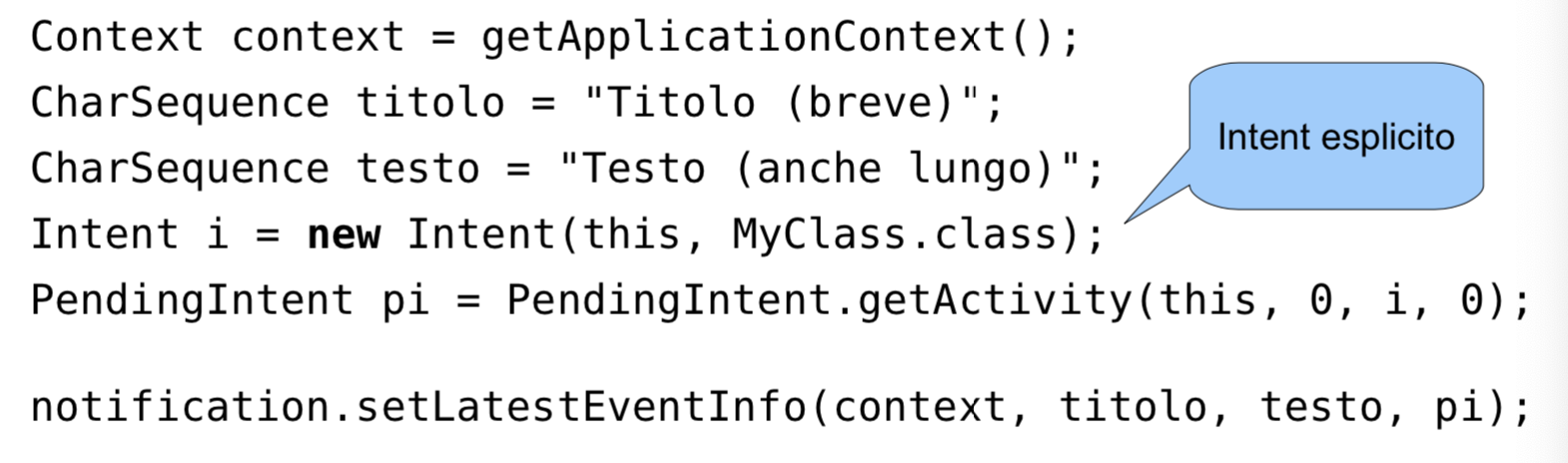
nm.notify(id, notification);

id – un nostro intero che serve a identificare la notifica

Spesso si vogliono dare più̀ informazioni quando l'utente “apre” la barra delle notifiche per avere dettagli, vorremmo anche aggiornare i dati di una notifica già̀ sottomessa 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.



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 una notifica già esistente:

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

Ad esempio arriva una nuova email e si aggiorna il conteggio mail “nuove”, 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 ottenuto il builder (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 aync (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.

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

Snoozing: consente all’utente di posporre una notifica per un certo tempo

Dismissal: la cancellazione ad opera dell’utente è ora distinta da quella via programma

Aspetto grafico: si possono impostare colori e stili grafici distinti senza produrre un’intera view

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

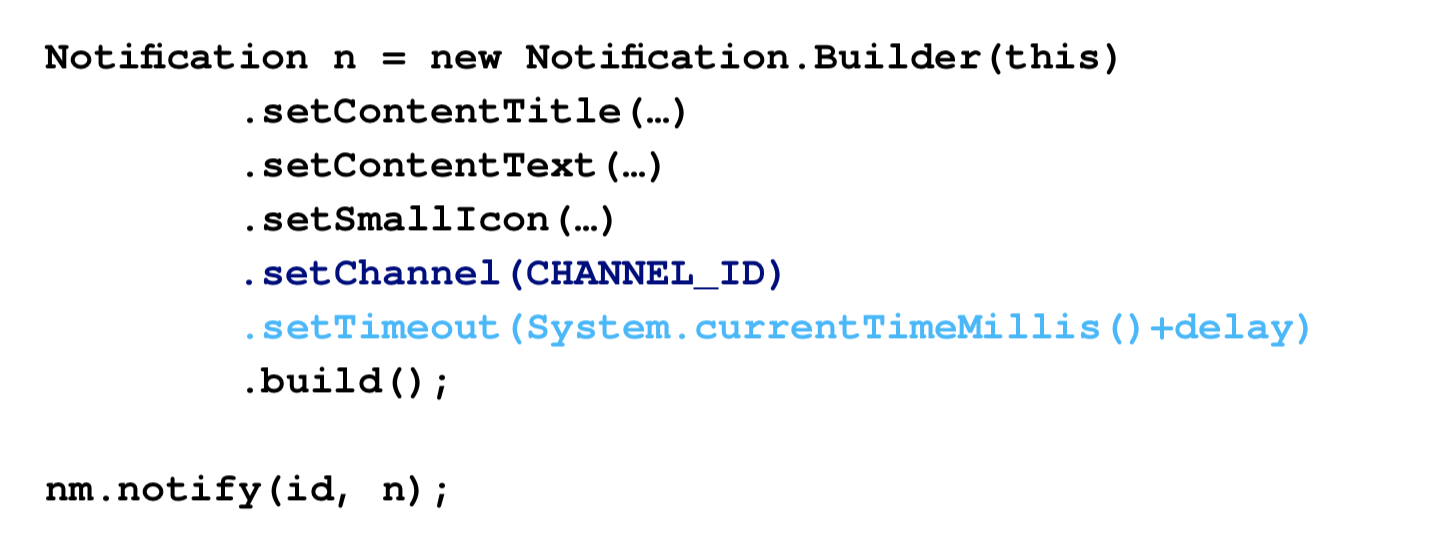
Priorità

Segnale sonoro, luminoso (LED), vibrazione

Se le notifiche vadano mostrate sul lock screen

Se le notifiche vadano mostrate in modalità Do Not Disturb

L’utente (non la app) definisce queste proprietà



In Android Q, Google ha aggiunto un servizio rimpiazzabile di gestione delle notifiche ancora più semplice, un NotificationAssistantService può ricevere le notifiche postate 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:

AlertDialog richieste di decisione

ProgressDialog stato di avanzamento di un task

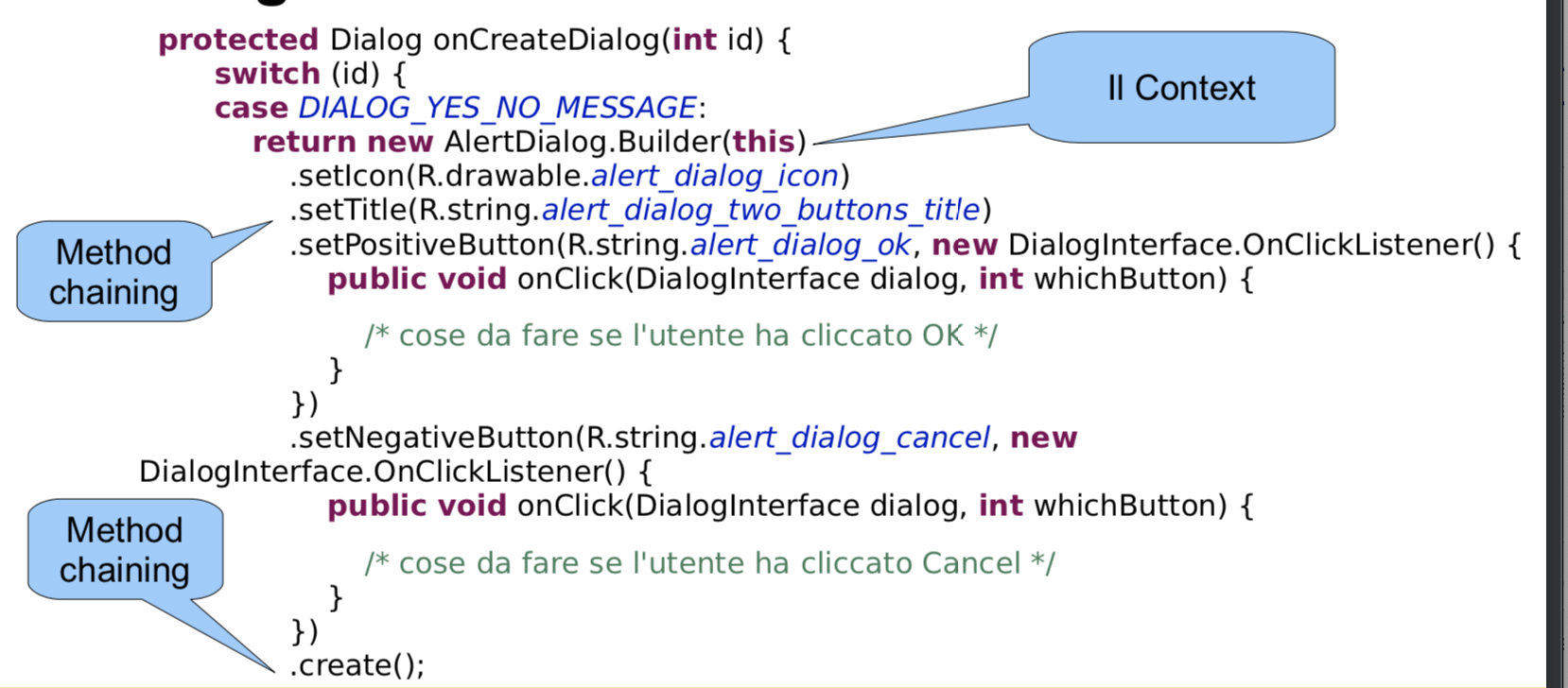
DatePicker e TimePicker input date e orari

Mentre un 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).

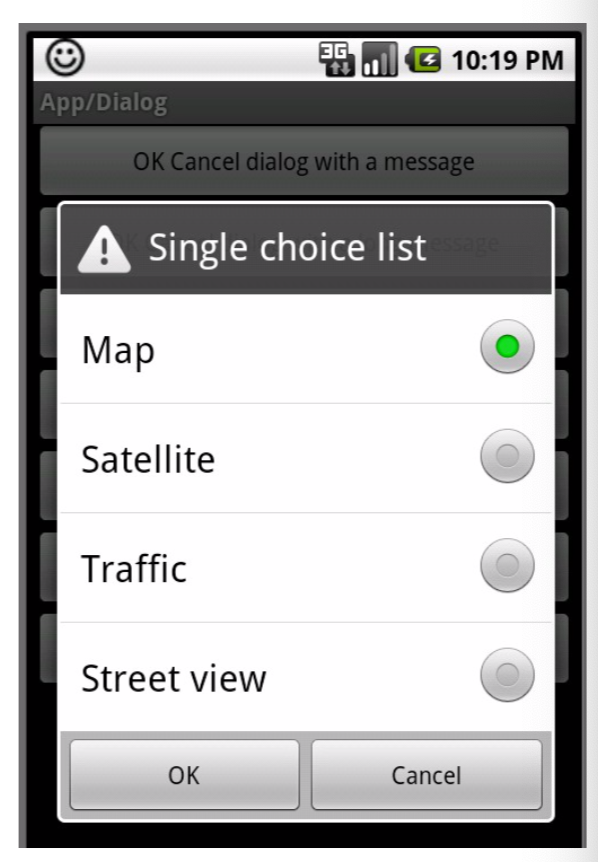
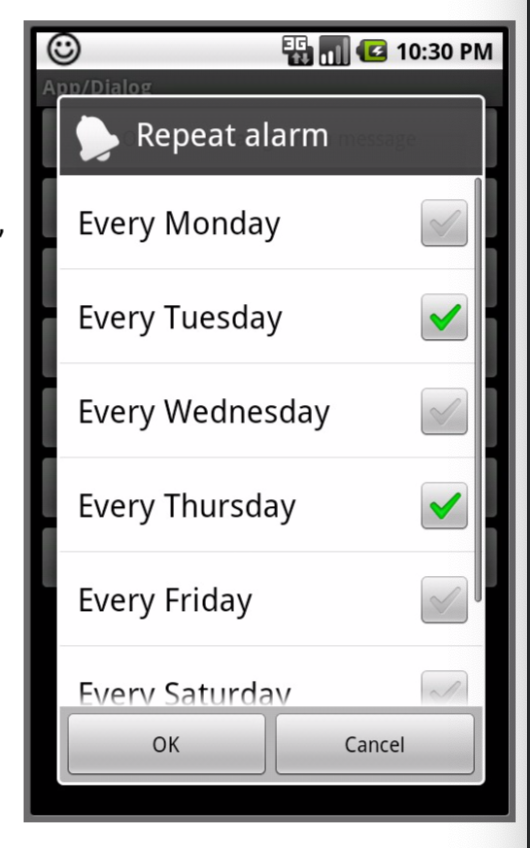
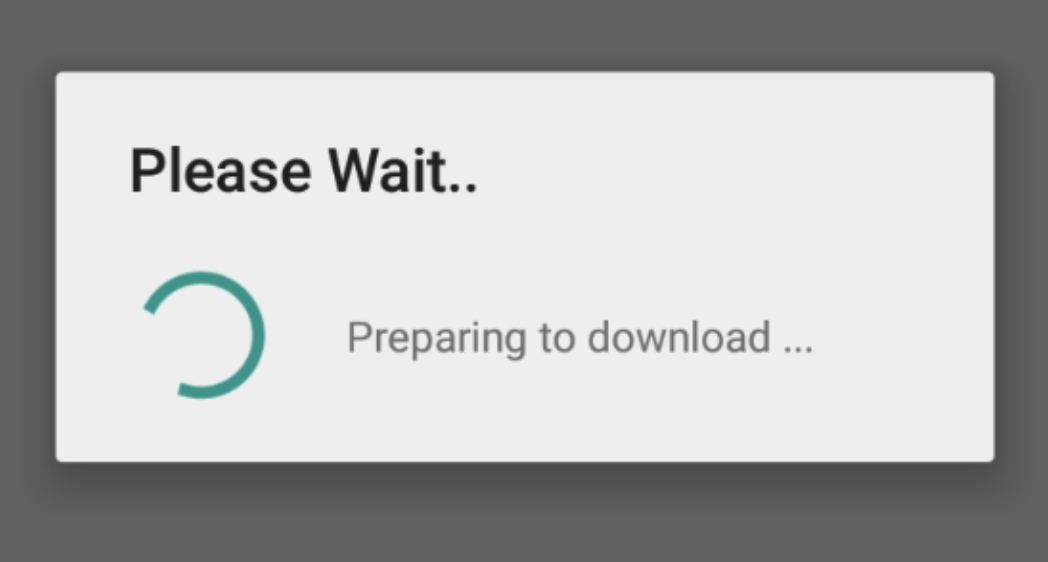
Prima di aprire il dialog, il sistema chiama onPrepareDialog(id, dialog) dell'Activity dove si possono cambiare i contenuti del dialog.

È sempre 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(). È consigliato usare un DialogFragment.

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



Ci sono svariati tipi di dialog, Dialog Single choice, MultiChoice e DialogProgress (quest’ultimo sconsigliato da lollipop in poi, spinner circolare di caricamento con i vari metodi come setProgress())

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 un loro salvataggio dello stato (onSaveInstanceState() con passaggio del bundle) I fragment possono avere un layout definito in XML oppure è possibile usare un FragmentManager. sono fornite 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 sottoclasse facendo obbligatoriamente override del metodo onCreateView().

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 carino usare le transizioni: 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. E' possibile associare una transizione grafica (predefinite del sistema o custom-file xml) alla transazione. Per specificare la transizione da usare, si chiamano altri metodi della transazione setTransition(int style) con intsyle = TRANSIT\_NONE, TRANSIT\_FRAGMENT\_OPEN, TRANSIT\_FRAGMENT\_CLOSE.

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

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()/onRestoreInstance() 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 comunication, 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.



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, 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 vogliano memorizzare preferenze/settings 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, rappresenza 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 ecc… 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 (commit atomico) coalescing (viene notificato una sola volta sola per l’intero insieme di modifiche). Le modifiche alle preferenze vengono eseguite tramite un editor che offre medoti del tipio “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)

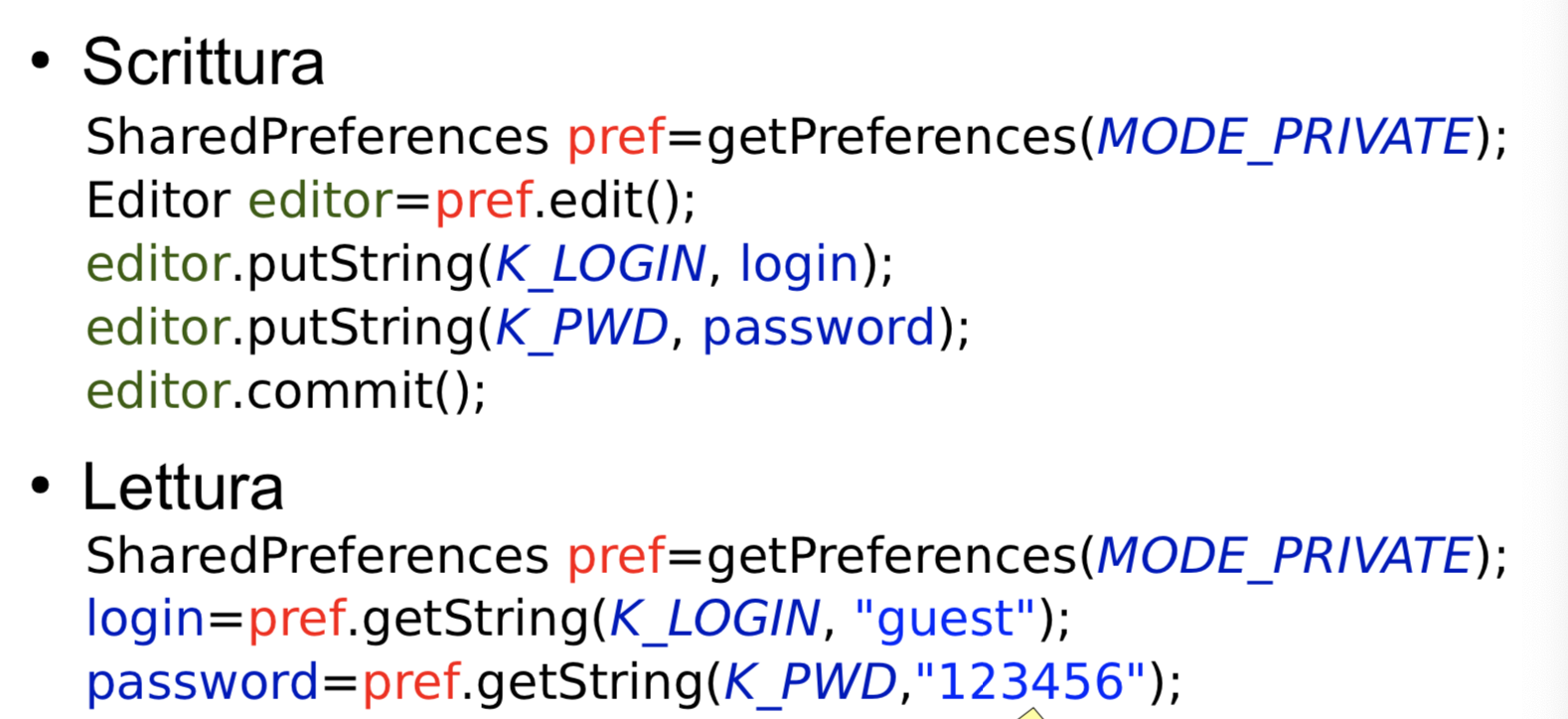
apply() : aggiunge la tabella dell'editor a quella delle preferenze in memoria, e schedula la scrittura asincrona del risultato su disco (Meno sicura, non verifica gli errori, più efficiente)

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:



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!

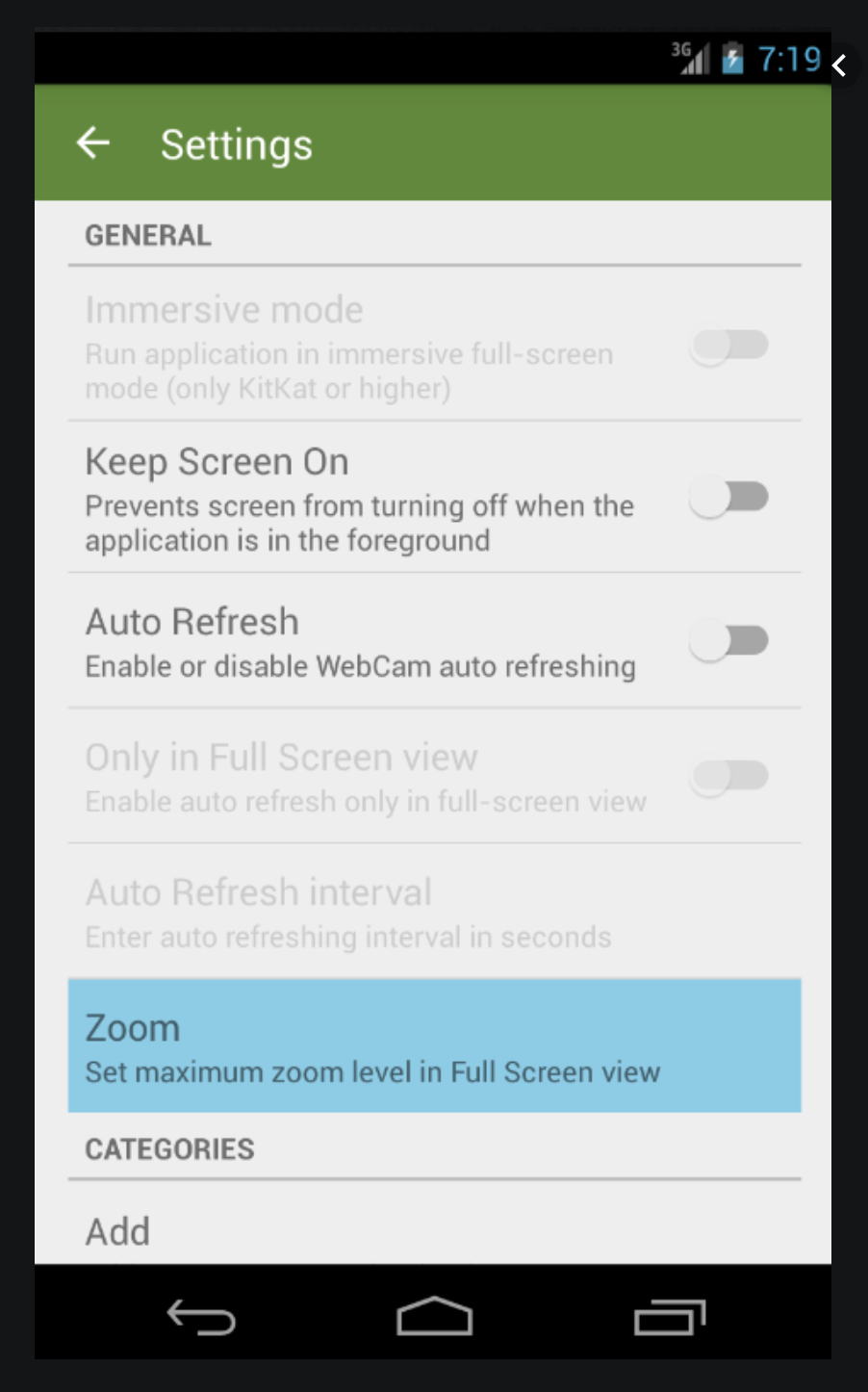
PREFERENCE SCREEN

Quando siamo a conoscenza di quali preferenze servono, si può costruire la GUI corrispondente in maniera sistematica. 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 PreferenceAtivity 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. La definizione del Fragment è del tutto standard:  
public class TestPreferenceFragment extends PreferenceFragment {…}.

ACCESSO AL FILE SYSTEM

Android è uno strato “sopra” Linux, il file system vero e proprio è quindi gestito interamente da quest'ultimo. (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)

Ecc…

e altri metodi per gestire le directory

File getFilesDir() – restituisce il path alla base dir

File getDir(nome, modo) – apre o crea una sottodir

File getCacheDir() – restituisce il path alla cache dir

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 – tutto ok!

Environment.MEDIA\_MOUNTED\_READ\_ONLY – solo lettura

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.

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()!

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

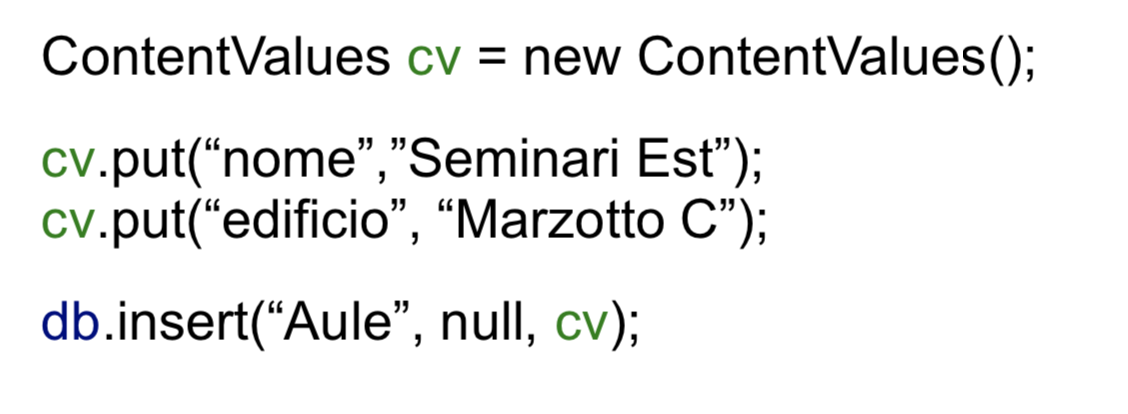
Il metodo più generale è db.execSQL(sql), esegue i comandi SQL passati (come stringa) sql può contenere qualunque comando, purché non debba restituire nulla (il metodo è void). In particolare, può eseguire CREATE TABLE e simili. NON può eseguire SELECT, può eseguire UPDATE, ma non restituire il numero di record modificati.

Il costo di compilazione di ogni istruzione SQL non è (affatto) trascurabile

SQLite fornisce una modalità alternativa, in cui anziché passare una istruzione SQL, si invocano specifici metodi:

delete(tabella, where, args)

insert(tabella, nullcolumn, valori)



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 servirce per riempire una listview ad esempio. Quindi 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 avessere 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.

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

getWriteableDatabase() – 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().

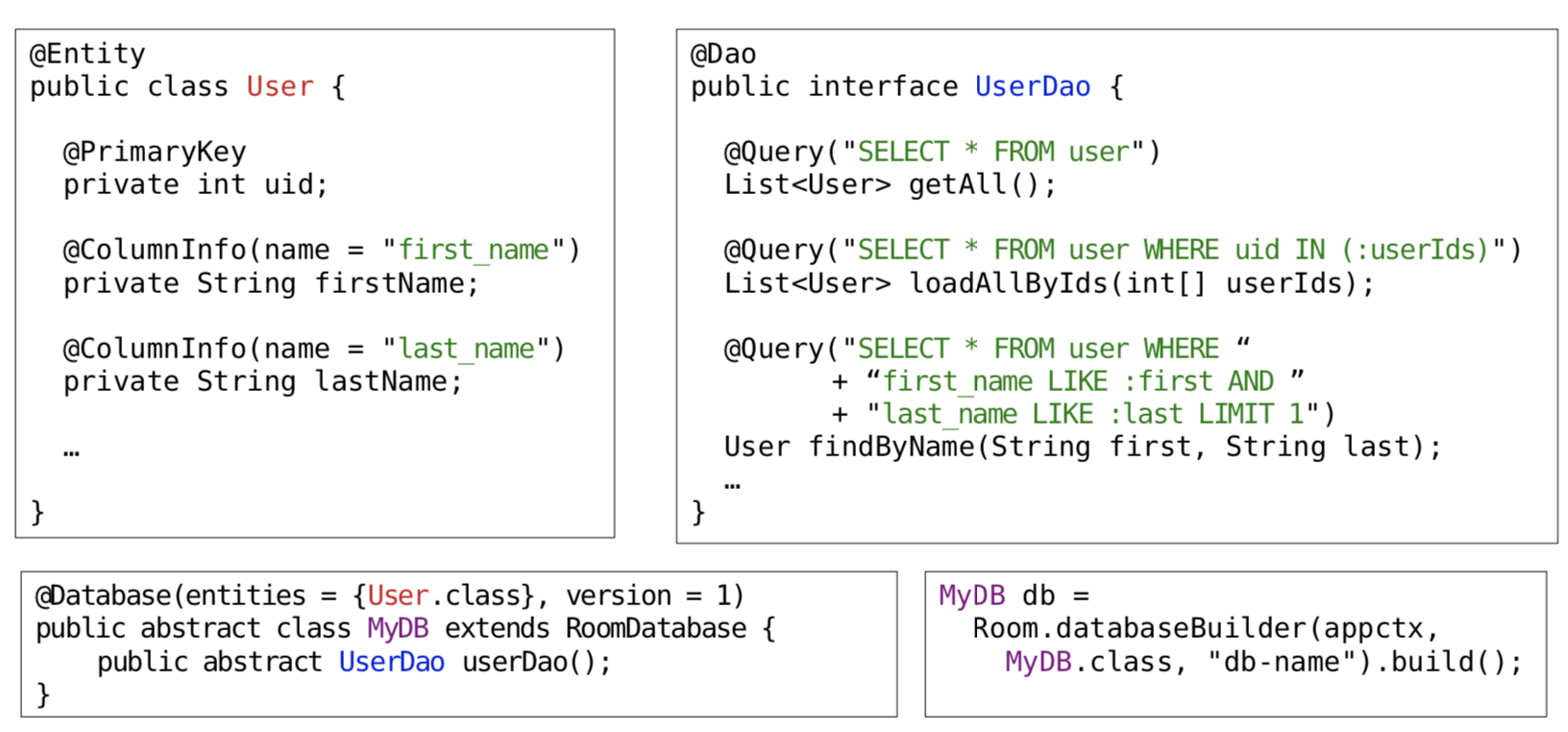
Se non ci sono stati errori, il DB viene restituito al chiamante.

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.

JETPACK ROOM

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



CONTENT PROVIDER

Abbiamo visto che le applicazioni Android possano 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: Due aspetti: 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, mai eseguirla nel thread della UI, la classe CursorLoader viene in aiuto caricando asincronamente i risultati di una query (vedremo dopo).

Leggere i dati da un ContentProvider:

Le URI usate nei content provider hanno un formato noto:

content://media/internals/images

content: schema

media: identifica il provier (authority)

internals/images : è il path

La classe android.net.Uri fornisce metodi di utilità per costruire e convertire URI. Ad esempio

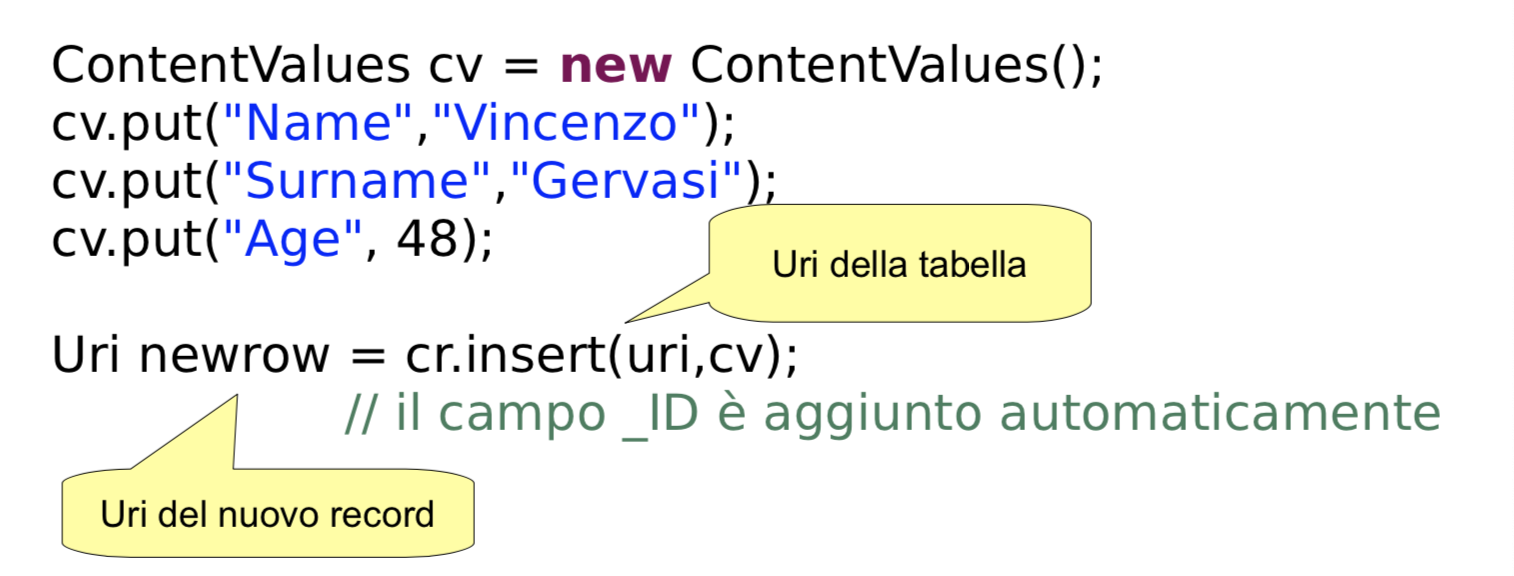
Uri Uri.parse(String s) data una stringa s, costruisce l'Uri relativa.

La classe Uri rappresenta una URI immutable, è efficiente, ma poco flessibile e fa poca validazione.

La classe Uri.Builder è un costruttore di URI progettato per manipolare URI 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.



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

Garantire che chi usa un ContentProvider “lo conosca” a fondo

(Il nome del permesso fa un po' da “password”)

DEFINIRE UN CONTENT PROVIDER

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. Normalmente viene istanziato quando un ContentResolver deve gestire un'URI la cui authority corrisponde alla nostra, viene chiuso quando non è più necessario.

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.

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 filesystem 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 fs di Linux.

Usando contents://file-provider/ si applicano i permessi di android. E’ possibile creare un intent che fa riferimento a un URI sul file provider ed è possibile importare i flag di permission lettura/scrittura.

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 una classe, metodo o package, dinamicamente può essere eseguito da un thread, da un processo ecc..

Su android un processo ha spazio degli indirizzi isolato.

Come in java l’oggetto thread rappresenta un thread della JVM ma non lo è. Finchè 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:

Comando: 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.

Dichiarazione: synchronized tipo m(arg) { blocco } = Prova ad acquisire il monitor dell'oggetto (o della classe) a cui appartiene il metodo di istanza (statico)

I costrutti synchronized offrono un modo per realizzare la mutua esclusione e per serializzare l'accesso da parte di diversi thread. Particolare cura va posta nel proteggere le strutture dati condivise fra più thread! Si possono usare le varianti “protette” delle collezioni.

#1 Mai usare thread della UI per operazioni lunghe

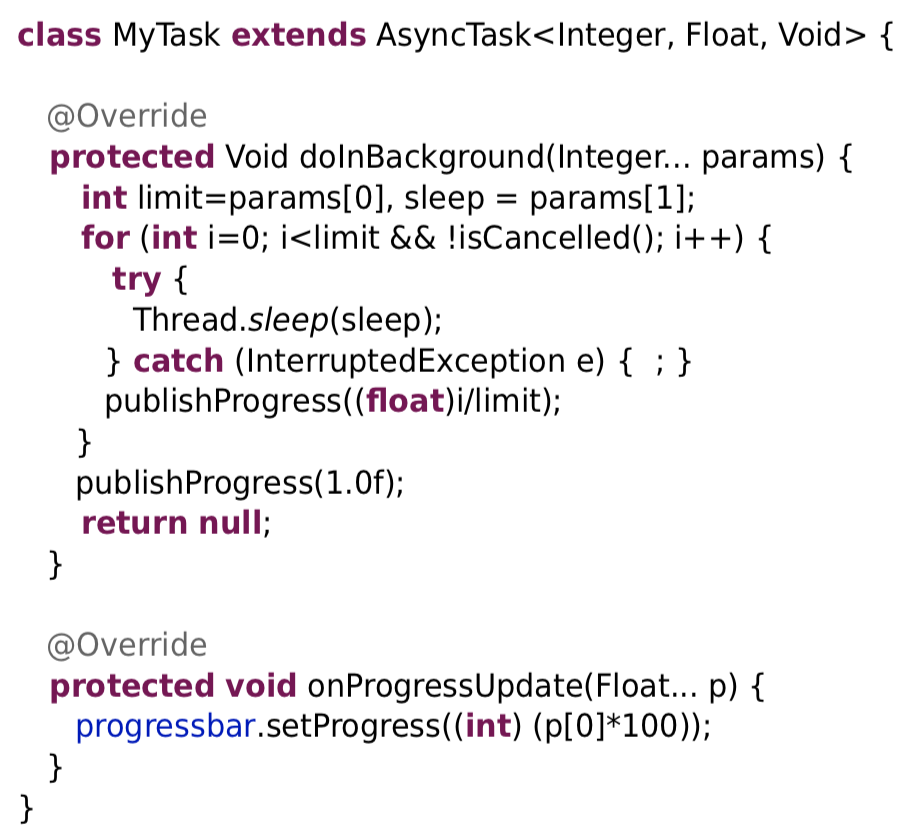
#2 Mai usare un thread diverso dal thread UI per aggiornare la UI.

PROBLEMA: Come posso fare se serve una operazione lunga che deve aggiornare la UI? Es.: accesso a DB, accesso alla rete, calcoli “pesanti”. Creare nuovi Thread mi aiuta per la regola #1, non per la #2

Il caso più comune è quando il thread UI deve far partire un task (lungo), il task deve aggiornare la UI durante lo svolgimento Il task deve fornire il risultato alla UI alla fine.

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().



La classe Activity offre “void runOnUiThread(Runnable r)”

Può essere chiamato da un thread non-UI, il runnable 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…

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).

In effetti, tutte le volte che abbiamo detto:

“dopo la richiesta il sistema, con suo comodo, in qualche punto del futuro, farà la tale operazione”

si intendeva:

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 una classe in grado di scaricare una immagine dal web e inserire la bitmap corrispondente in una ImageView

Scaricamento su thread separato

Inserimento nella view su thread.

Usiamo l’asynctask. Normalmente tenere un riferimento (=puntatore) a un oggetto impedisce al garbage collector di disallocarlo. Il risultato è che se l’activity è stata chiusa, la memoria occupata non può essere liberata. Per evitare questa situazione usiamo i weakreferences, esso si comporta come un riferimento ma non impedisce al garbage collector di disallocare la memoria (sempre se non ci sono altri riferimenti non weak) in caso di disallocazione al get restituisce null. Il download della foto può essere eseguito usando la classe URL e il metodo downloadBitmap oppure usando la classe AndroidHttpClient. Android offre un servizio di sistema per il download di risorse via http, il servizio è asincrono, gestisce lo scaricamente, la notofica di progress e segnala il completamento all’applicazione. Come per tutti i servizi di sistema, si può usare la getSystemService() DownloadManager dm = getSystemService(DOWNLOAD\_SERVICE); Una volta ottenuto il servizio, si possono accodare download con long id = dm.enqueue(request);

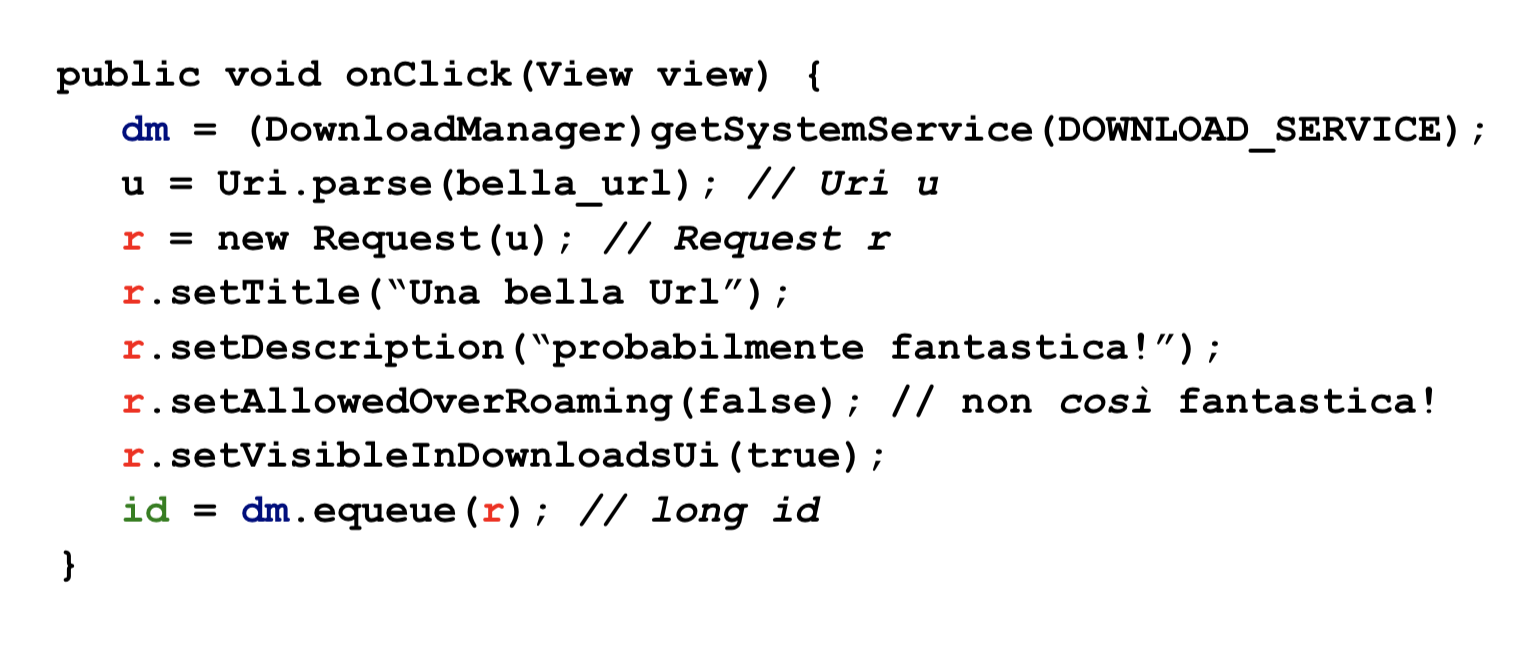
E al termine, essere avvisati da un broadcast ACTION\_DOWNLOAD\_COMPLETE

L'app dovrà registrare un BroadcastReceiver. La classe DownloadManager.Request incapsula una richiesta di scaricamento. Contiene tutti i dettagli del caso, fra cui L'URI da scaricare (obbligatoria!)

Dove mettere il file locale (per default, spazio shared) Se visualizzare una notifica o no durante il download Un titolo e una descrizione (per le notifiche, se ci sono)

Restrizioni sulle reti da usare: solo in wi-fi, anche in cellulare, anche in roaming, ecc.

Immaginiamo di avere un pulsante “download”...



Registriamo un receiver

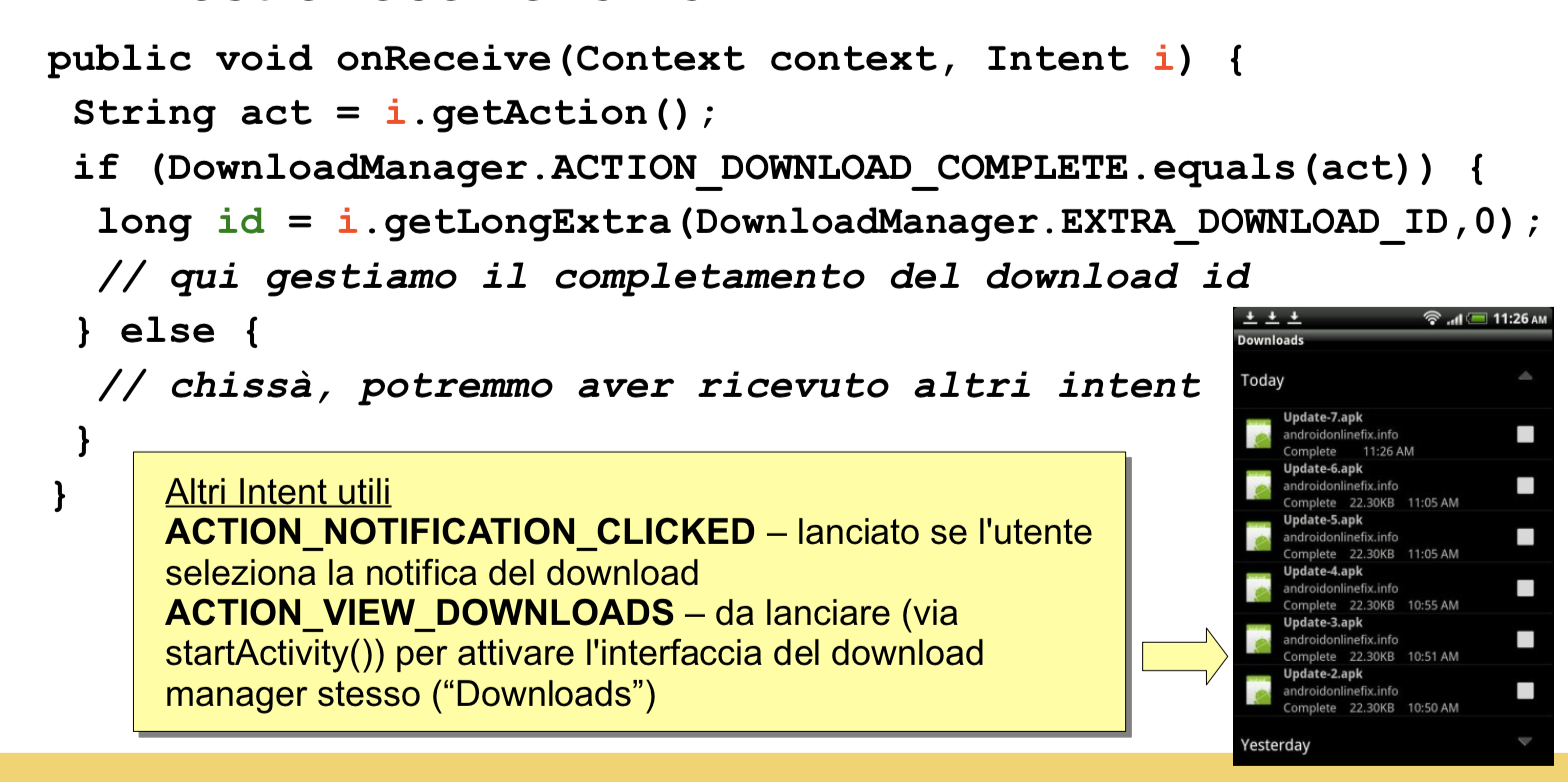
registerReceiver(receiver,

new IntentFilter(DownloadManager.ACTION\_DOWNLOAD\_COMPLETE)

);

Qui ha senso usare la registrazione dinamica

Registriamo il receiver quando avviamo un download Lo de-registriamo al completamento, In altri casi potrebbe essere meglio registrarlo staticamente (AndroidManifest.xml).



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

Il DownloadManager mantiene un ContentProvider con le informazioni sui download

Le applicazioni possono

Richiedere di aggiungere le informazioni sui propri download al provider

Fare query sul provider per avere informazioni sui file scaricati

La classe DownloadManager.Query fornisce metodi di utilità per le query più frequenti

CURSORI ASINCRONI

La classe astratta loader specifica un protocollo generico per il caricamento asincrono dei dati.

Di tratta di componenti attivi che si occupano di aggiornare quando vengono aggiornati i dati sottostanti. Un loader è identificato con un ID univoco, se non esiste già un loader con quell’ID viene creato e inizializzato se invece esiste già viene ripristinato il suo stato precedente dal bundle.

ASYNCPLAYER

Molte classi di sistema gestiscono il multithreading completamente al loro interno

AsyncPlayer è una di queste, riproduce file audio, ovviamente in background Il thread non si vede, ma c'è. L'uso da parte delle applicazioni è banale, se sono richieste capacità più avanzate, c'è tutta 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 estendo BroadcastReceiver, si pubblica il componente in Android Manifest.xml con i relativi intent filter. I BR hanno un ciclo di vita semplicissimo, viene instanziato quando si ha bisogno, l’intent viene passato al suo metodo onReceive(), al ritorno da onReceive() l’oggetto viene rilasciato. 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 è una TRAGGEDIA. Molti servizi di sistema che inviano delle notifiche broadcast finiscono per essere gestiti da BroadcastReceiver

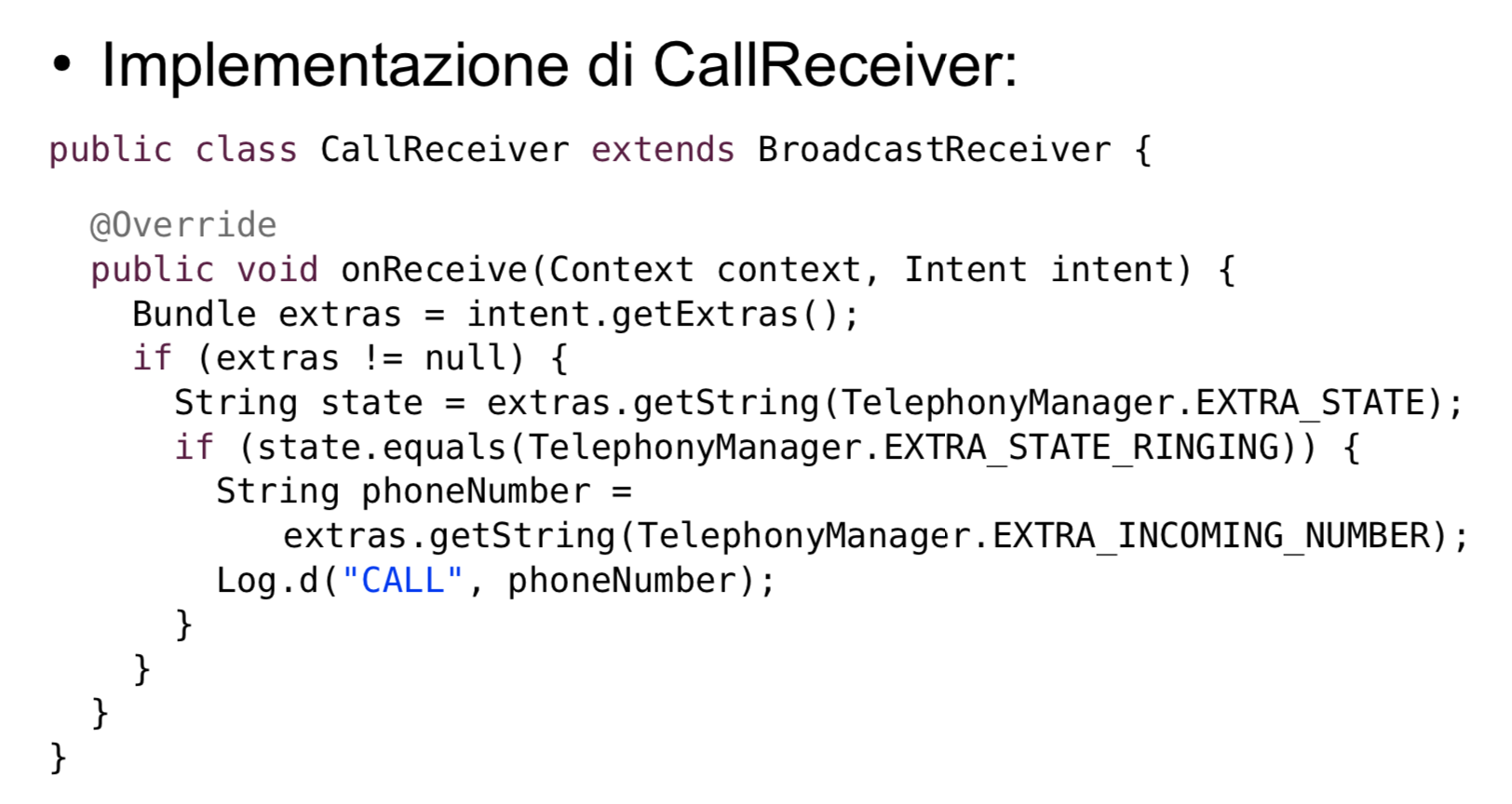
Anche un'activity può essere attivata da un Intent

Ma è molto brutto per eventi asincroni (si interrompe l'utente)

Brutto: arriva un SMS, e parte l'Activity di messaggistica

Meglio: arriva un SMS, parte un Broadcast Receiver, il quale posta una notifica nella Notification Area, che poi quando selezionata dall'utente fa partire l'Activity di messaggistica.





È anche possibile registrare e deregistrare dinamicamente un BroadcastReceiver

In questo modo, l'applicazione riceve broadcast solo quando il receiver è registrato

public abstract Intent registerReceiver (BroadcastReceiver receiver, IntentFilter filter)

public abstract void unregisterReceiver (BroadcastReceiver receiver)

Alcuni Intent inviati in broadcast possono essere definiti sticky

Dopo essere stati inviati in broadcast a tutti i receiver del sistema il cui IntentFilter corrisponde all'Intent sticky, rimangono “vivi”

Se successivamente al broadcast si registra dinamicamente un nuovo BroadcastReceiver che corrisponde,

L'Intent sticky viene inviato normalmente al nuovo receiver 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 intent sticky, le app devono avere il permesso BROADCAST\_STICKY

In effetti, sono tipicamente usati (solo) dal sistema

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 di questo tipo inviato

Nella maggior parte dei casi, le applicazioni saranno interessate a ricevere i broadcast di sistema

Quelli di sistema sono ben noti. 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

public abstract void sendBroadcast (Intent intent): invia i a tutti I BroadcastReceiver corrispondenti

startActivity(i) → invia i a una Activity corrispondente

L'intent è lo stesso, i meccanismi di dispatch sono diversi!

La chiamata a sendBroadcast() è asincrona

Ritorna immediatamente al chiamante, nel frattempo il sistema, con calma, manda gli Intent.

public abstract void sendBroadcast (Intent intent, String receiverPermission): Invia solo a chi ha dichiarato un certo permesso.

Invio serializzato e ordinato

Normalmente, i BroadcastReceiver di app diverse vengono eseguiti concorrentemente

È possibile chiedere invece l'invio serializzato e ordinato (in base al valore di android:priority del receiver)

public abstract void 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: public final void abortBroadcast(),Per avere dei risultati, si invia l'intent con

public abstract void 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

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()

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

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!

Per compiti di temporizzazione pura, meglio usare altri costrutti

es.: 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.

Gli allarmi pendenti vengono annullati con lo shutdown del dispositivo

Molto poco adatto, per esempio, per una sveglia!

La ri-abilitazione 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

Gli allarmi dell’AlarmManager possono essere posticipati

Alla successiva maintainance window Non viene fatto lo scan delle reti wi-fi

Non vengono eseguiti i sync adapter e i task del JobScheduler

Sul JobScheduler diremo di più la prossima lezione

È possibile chiedere esplicitamente di impostare degli allarmi critici che superano il Doze:

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

am.setExantAndAllowWhileIdle(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.), vd. farm, pipeline, map/reduce, e passare mappe chiavi/valori fra task

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

Un task da eseguire è definito da una istnaza di Worker, come al solito faremo la extends della classe, con un solo metodo da implementare doWork().

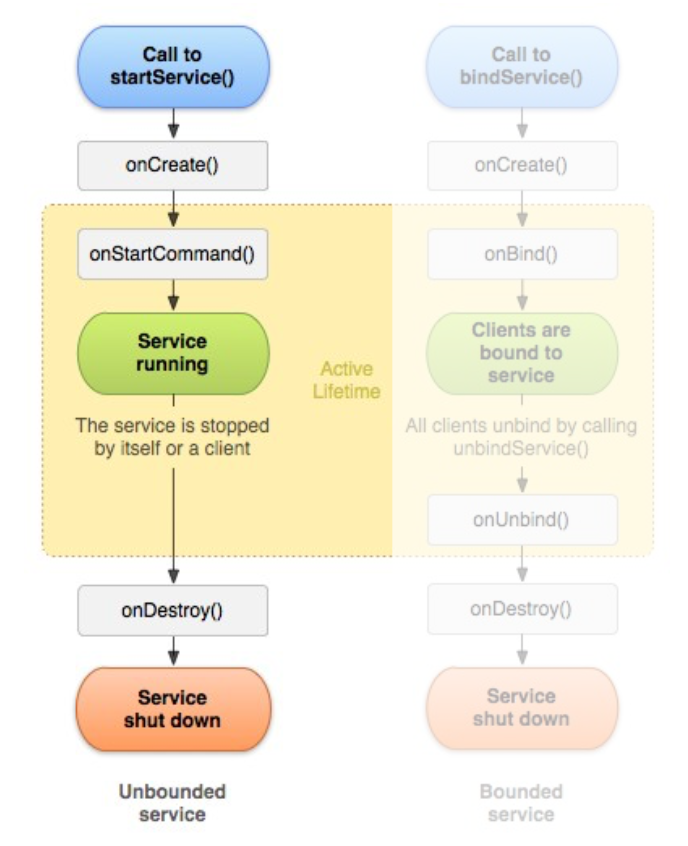
La richiesta di esecuzione di un task (ovvero, di un Worker) è definita da un’istanza di una sottoclasse di WorkRequest

Il sistema fornisce due sottoclassi predefinite:

OneTimeWorkRequest – esegui il task una volta sola PeriodicWorkRequest – esegui il task ripetutamente. Per finire, è possibile specificare quali vincoli (di sistema) devono essere verificati affinché un lavoro sia eseguibile

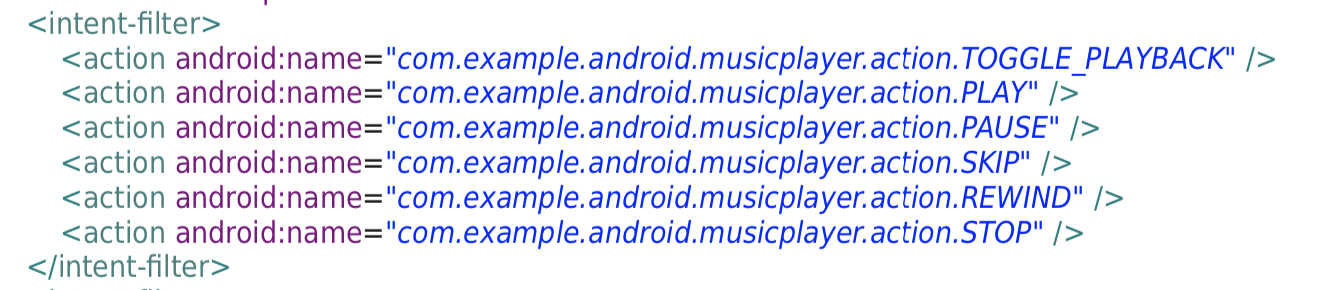
Vincoli rappresentati come istanze di Constraints.

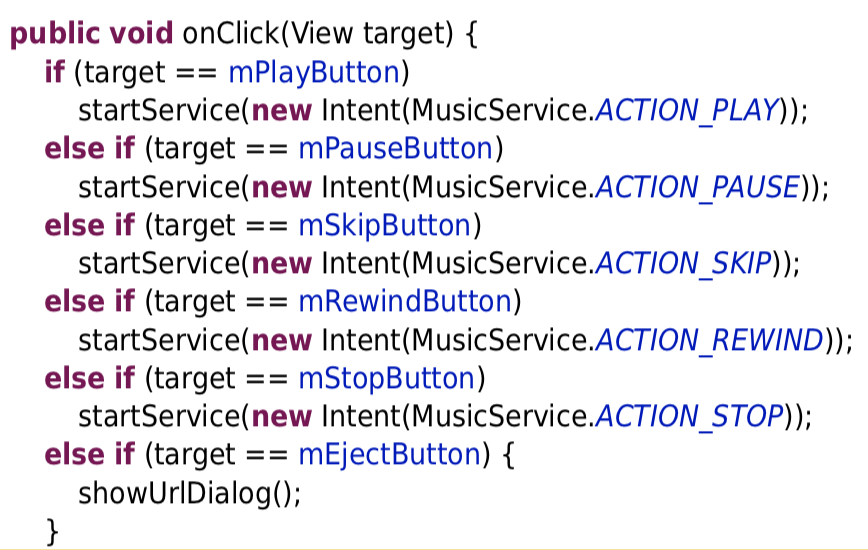
SERVICE

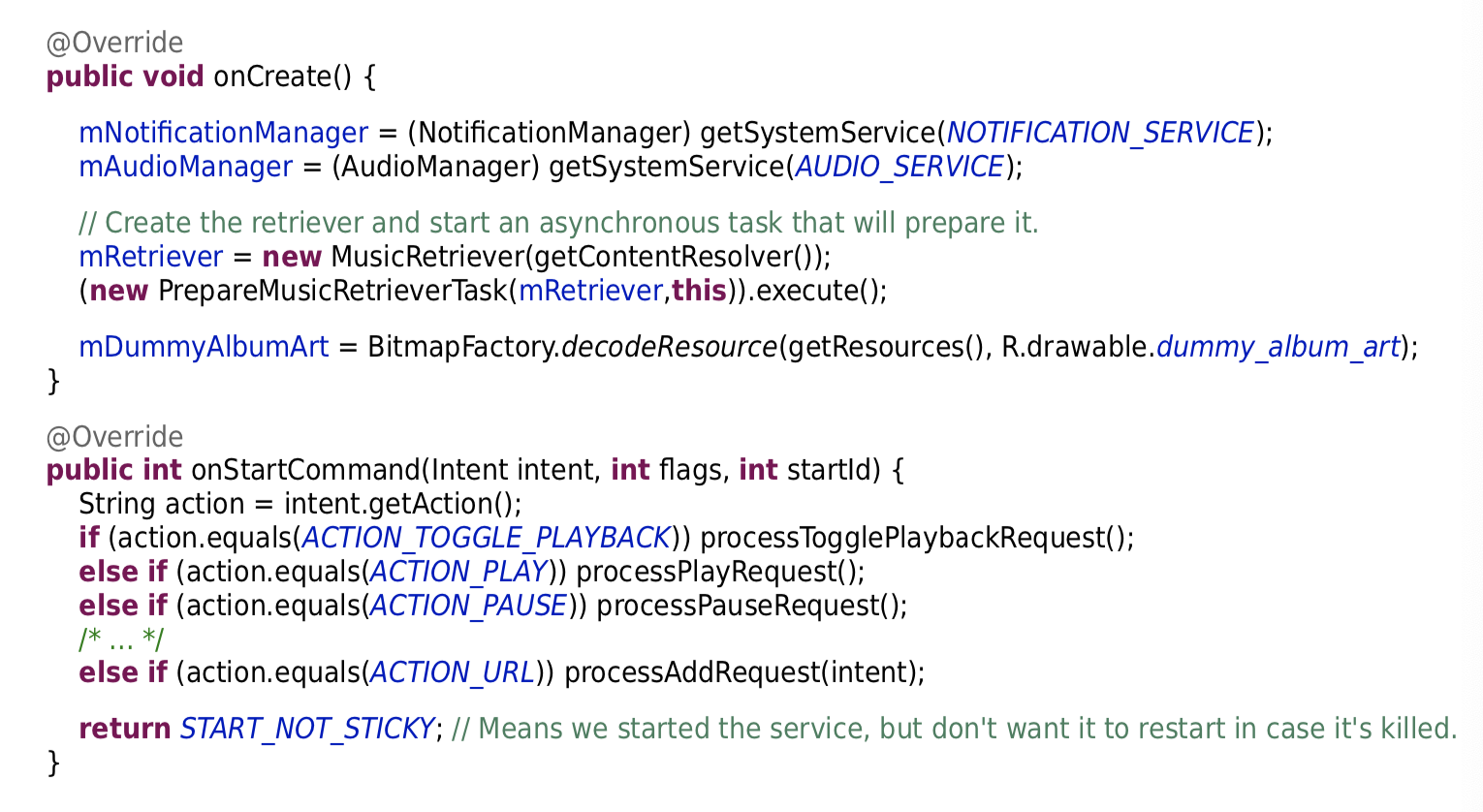
Dopo le Activity, i Content Provider e i BroadCastReceiver i Services sono il quarto tipo di componente di una applicazione. Un Service è un oggetto che può eseguire del codice senza disporre di una interfaccia utente, opera sempre in background rispetto alla UI, ma il thread dove viene eseguito è quello 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. 

Nel caso di Service started una call a startService() chiede se il service sia attivo, se è attivo non siccede nulla altrimenti viene instanziato e inizializzato. (non esiste il nesting). Nel caso di Service bound, il sistema tiene traccia di quanti clienti al momento usano il Service. Quando il conteggio raggiunge 0, il Service può essere dismesso. Le connessioni sono permanenti. Per avviare un service basta inviargli un Intent e avviarlo con startService(intent). Il sistema istazierà il service e chiamerà la sua onCreate(). Poi passerà l’intet 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.)

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! 





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

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

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

Può anche rappresentare un'Activity, un BroadcastReceiver o un ContentProvider: 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

Il componente che necessita di un Service prepara un Intent che lo identifichi

E mette dentro, come data o extra, tutto il necessario

● Il componente chiama startService(intent)

● Il sistema “trova” il Service giusto

Se necessario, lo lancia → onCreate() Comunque, onStartCommand(intent, flag, id)

● Il Service serve l'intent

Eventualmente, invia una risposta al componente chiamante – Perché lo conosce, o via un PendingIntent passato come extra

● Il Service chiama stopSelf(id)

Oppure rimane indefinitamente in giro, finché qualcuno non chiama

stopService()

Quasi sempre, è utile che un Service usi uno o più thread separati per servire le richieste

Ricordate: i metodi del ciclo di vita di un Service sono eseguiti dal thread UI (come per gli altri componenti)

IntentService è una sottoclasse di Service che:

Serve le richieste in un thread separato

In caso di richieste multiple, gestisce una coda

Nota: non gestisce richieste in parallelo; se vi serve il parallelismo, dovrete implementare voi 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() è un metodo astratto che voi dovete

implementare in una vostra sottoclasse di

IntentService

protected abstract void onHandleIntent(Intent intent);

In definitiva:

Create una sottoclasse di IntentService

Fate overload di onHandleIntent()

Il vostro codice sarà eseguito da un thread separato

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, tutto bene Se 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. Tiè.

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

Cross-process, con AIDL

Struttura analoga a CORBA, RMI, RPC ecc. – non lo vediamo

Il servizio 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

onServiceConnected(ComponentName n, IBinder binder) onServiceDisconnected(ComponentName n)

flags: precisa la gestione della priorità del servizio, per esempio BIND\_IMPORTANT o BIND\_NOT\_FOREGROUND; BIND\_AUTO\_CREATE è un buon default

La bindService(intent, conn, flag) causa una chiamata alla onBind(intent) del Service (e forse una onCreate())

bindService() è void e termina subito: il binding poi è asincrono

● La onBind(intent) restituisce un nostro oggetto binder che implementa

l'interfaccia IBinder

Spesso è una sottoclasse di Binder, e fornisce un metodo getter per il Service stesso

In teoria, può implementare un'interfaccia “pubblica” separata per il nostro Service

● 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)

Anche qui, abbiamo un rischio sicurezza se usiamo un Intent implicito in bindService()

Non possiamo sapere quale Service risponderà Ma qui è peggio rispetto a prima

Perché poi 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 Come se già non fosse complicato di suo

Meglio, in generale, scegliere uno stile e mantenerlo

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

LEZIONE 18