

Mobile computing (sviluppo Android)

D'Angelo Carmine

E

Vitale Emanuele

Introduzione ad Android

Architettura sistema Android

Si basa su un kernel linux:



Spiegazione dei livelli:

1. **Livello kernel:** fornisce i servizi di base del sistema operativo, come : filesystem, gestione della memoria e dei processi, gestione dell'interfaccia di rete e drivers per le periferiche. Android ha anche dei servizi specifici e sono: la gestione della batteria, gestione della memoria condivisa, low memory killer (quando c'è poca ram disponibile chiude i processi) , interprocess commincation (comunicazione tra processi) e ecc..
2. **Livello astrazione hardware (HAL):** ci sono le interfacce standard per esporre le capacità hardware ai servizi di livello superiore (audio, bluetooth, fotocamera, ecc.).
3. **Android runtime (ART e Dalvik VM):** Android Runtime è una VM specifica per i sistemi Android, ha CPU meno veloci rispetto ad un pC, meno RAM e ha una batteria limitata. ART

accetta da 5.0 livello API 21 mentre Dalvik API < 21. Le app che funzionano su ART funzionano anche su Dalvik il contrario no.

4. **Librerie native:** molte componenti android necessitano di librerie native (webkit, libc, OpenGL ES, ecc.).
5. **Application framework:** tutte le funzionalità del S.O. Android vengono esposte tramite un API:
 - **View System:** fornisce gli elementi di base per le interfacce utente (icone, testo, bottoni, ecc).
 - **Content providers:** per accedere a dati di altre app, per esempio ai contatti della rubrica.
 - **Package manager:** gestisce l'installazione delle app sul dispositivo mobile
 - **Activity Manager:** gestisce il ciclo di vita delle applicazioni, permette di passare da un'applicazione ad un'altra.
1. **Applicazioni:** sono le applicazioni già presenti nel cellulare, e quelle che saranno installate nel futuro.

Java in Android

Le app in Android sono scritte in Java, la libreria fornisce molte classi pronte all'uso:

- classi di base: java.*, javax.*
- classi per le app: android.*
- Internet/web services: org.*
- Unit testing: junit*.

Le app sono:

- Scritte in Java
- Compilate in file Java Bytecode
- Un tool, DX, trasforma i file bytecode in un singolo file Dex Bytecode (classes.dex)
- Il file classes.dex contiene anche tutti i file di dati necessari e viene installato sul target device
- ART Virtual Machine esegue il file Dex

Directory Android

1. **Manifesto:** informazioni generali sull'app (permessi, attività, icona)
2. **Java:** file sorgenti
3. **res, risorse:**
 - **Drawable:** contiene tutto ciò che è disegnabile (immagini, file xml per specifiche grafiche).
 - **Layout:** descrizione dell'interfaccia grafica dell'app.
 - **Values:** contiene dei valori numerici (conviene metterli i valori numerici in un file xml inserito in questa directory, per potervi accedere dalle classi java, così se un valore è presente in più classi basta cambiarlo una volta solo).
 - **Menu:** per i menù pop up
 - **Mipmap:** serve a gestire immagini di varie dimensioni, cioè stesse immagini ma a diverse risoluzioni.
1. **Gradle:** contiene files che tengono traccia delle dipendenze per generare l'apk.
2. **Informazioni:** dipendenze da altro codice

Emulatore vs Real device

- **Real device:** Android Virtual Device Manager veloce, facile gestire l'input (es. rotazioni display) e l'esecuzione è reale. Attivare modalità sviluppatore e debug USB!!!
- **Emulatore:** lento (a volte molto), alcune operazioni sono difficoltose è comunque un "simulatore" e possono esserci dei bug. Facile creare situazioni particolari:
 - batteria scarica
 - arrivo di un messaggio

Supporto multi-lingue

Le app includono risorse che possono essere specifiche per una particolare cultura. Ad esempio, un'app può includere stringhe specifiche per la lingua che vengono tradotte nella lingua delle impostazioni internazionali correnti. È una buona pratica mantenere separate le risorse specifiche della lingua dal resto della tua app. Android risolve le risorse specifiche per lingua in base alle impostazioni locali del sistema. È possibile fornire supporto per diverse impostazioni locali utilizzando la directory delle risorse nel progetto Android. È possibile fornire qualsiasi tipo di risorsa appropriata per la lingua dei propri utenti. Per aggiungere supporto per più lingue, bisogna creare directory aggiuntive all'interno di res/. Il nome di ogni directory dovrebbe rispettare il seguente formato:

- **<res> values-b + <codice lingua>**, es: **res/values-b+it/strings.xml**

per le icone personalizzati invece:

- **<res> values-b + <codice lingua> [+<codice paese>]**, es: **res/mipmap-b+it+IT/country_flag.png**

Android carica le risorse appropriate in base alle impostazioni locali del dispositivo in fase di runtime.

In java basta scrivere il seguente codice:

```
// Get a string resource from your app's Resources
String hello = getResources\(\).getString(R.string.hello_world);
// Or supply a string resource to a method that requires a string
TextView textView = new TextView(this);
textView.setText(R.string.hello_world);
```

Sviluppo Android

Ogni widget dell' SDK Android è istanza della classe view o una sua sottoclasse.

Listeners: sono metodi degli oggetti della classe view, sono sempre in "ascolto" per entrare in azione quando si verifica un evento specifico, ad esempio un pulsante ha il metodo onClick che viene eseguito quando l'utente preme il pulsante.

Manifest.xml

Ogni progetto android deve avere un file AndroidManifest.xml (con esattamente questo nome) nella radice del set di origine del progetto. Il file manifest descrive le informazioni essenziali sulla tua app per gli strumenti di sviluppo Android, il sistema operativo Android e Google Play.

Tra le altre cose, è richiesto il file manifest per dichiarare quanto segue:

- Il nome del pacchetto dell'app, che di solito corrisponde allo spazio dei nomi del tuo codice. Gli strumenti di build di Android lo utilizzano per determinare la posizione delle classi durante la creazione del progetto.
- I componenti dell'app, che comprendono tutte le attività, i servizi, i ricevitori di trasmissione e i fornitori di contenuti. Ogni componente deve definire proprietà di base come il nome della sua classe Kotlin o Java. Può anche dichiarare funzionalità quali le configurazioni dei dispositivi che può gestire e filtri di intent che descrivono come il componente può essere avviato.

- Le autorizzazioni di cui l'app ha bisogno per accedere a parti protette del sistema o altre app. Dichiara inoltre qualsiasi autorizzazione che devono avere altre app se vogliono accedere ai contenuti da questa app.
- Le funzionalità hardware e software richieste dall'app che incidono su quali dispositivi possono installare l'app da Google Play.

Virtual device manager

Informazioni sugli AVD

Un AVD contiene un profilo hardware, un'immagine di sistema, un'area di memoria, skin e altre proprietà. Si consiglia di creare un AVD per ogni immagine di sistema che l'app potrebbe potenzialmente supportare in base all'impostazione `<uses-sdk>` nel manifest.

- **Profilo hardware:** Il profilo hardware definisce le caratteristiche di un dispositivo, cioè quelle di fabbrica. AVD Manager viene fornito precaricato con determinati profili hardware, come i dispositivi Pixel, ed è possibile definire o personalizzare i profili hardware secondo necessità. Si noti che solo alcuni profili hardware sono indicati per includere Play Store. Ciò indica che questi profili sono pienamente compatibili con CTS e possono utilizzare immagini di sistema che includono l'app Play Store.
- **Immagini di sistema:** Un'immagine di sistema etichettata con le API di Google include l'accesso ai servizi di Google Play. Un'immagine di sistema con il logo di Google Play nella colonna Play Store include l'app Google Play Store e l'accesso ai servizi di Google Play, inclusa una scheda Google Play nella finestra di dialogo Controlli estesi che fornisce un comodo pulsante per l'aggiornamento dei servizi Google Play sul dispositivo .
Per garantire la sicurezza delle app e un'esperienza coerente con i dispositivi fisici, le immagini di sistema con Google Play Store incluse sono firmate con una chiave di rilascio, il che significa che non è possibile ottenere privilegi elevati (root) con queste immagini.
- **Storage area:** L'AVD ha un'area di archiviazione dedicata sulla macchina di sviluppo. Memorizza i dati dell'utente del dispositivo, come app e impostazioni installate, nonché una scheda SD emulata. Se necessario, è possibile utilizzare AVD Manager per cancellare i dati dell'utente, in modo che il dispositivo abbia gli stessi dati come se fosse nuovo.
- **Skin:** Una skin dell'emulatore specifica l'aspetto di un dispositivo. Il gestore AVD fornisce alcune skin predefinite. Puoi anche definire la tua o usare skin fornite da terze parti.
- **AVD e caratteristiche dell'app:** Assicurati che la tua definizione di AVD includa le funzioni del dispositivo da cui dipende la tua app, come navigatore, fotocamera, memoria Ram, screen size, screen resolution, ecc..

Organizzazione file sviluppo android

- **cartella Manifest:** contiene il manifest.xml della nostra app.
- **cartella Java:** contiene le classi/activity scritte in Java che fanno parte dell'applicazione
- **cartella generatedJava:** le classi che puoi trovare qui sono tutte le classi di cui il progetto sarà costruito, comprese le librerie.
- **Cartella res:** contiene le resource directory, che si dividono nelle seguenti:
 - animator: contiene i file xml che definiscono le animazioni.
 - color: contiene i file xml che definiscono i colori.
 - drawable: contiene i file Bitmap o file xml che definiscono risorse drawable.
 - mipmap: file drawable per diversi tipi di avvisi, ad esempio un app che viene aperta da device con lingue diverse.
 - layout: contiene i file xml che definiscono il layout dell'interfaccia utente.
 - menu: definisce il layout per i menu.
 - raw: contiene file di tipo raw, come per esempio l'id di un elemento di layout es: R.id.name, la struttura di base è R.raw.filename.

- values: contiene file xml che contengono delle stringhe che possono essere usate nell'app.
- xml: contiene dei file xml che possono essere chiamati in qualunque momento nell'app.
- font: contiene i font da usare nell'app.

Layout

Definiscono l'aspetto grafico dell'interfaccia utente, sono anch'essi oggetti di tipo View. Si possono definire in due modi:

- Con un file XML (layout statico)
- in modo programmatico (layout dinamico).

Non sono mutuamente esclusivi, si possono usare in sinergia.

Vantaggi e svantaggi del layout statico e dinamico

Layout statico

- vantaggi:
 - Il vantaggio nell'utilizzo dei files XML è la separazione della presentazione dell'applicazione dal codice che ne controlla il comportamento.
- svantaggi:
 - una volta definito il layout nel file xml non è più possibile cambiare la disposizione degli elementi perché sono statici.

Layout dinamico

- vantaggi:
 - è possibile aggiungere o rimuovere elementi a runtime (facilmente adattabile)
- svantaggi:
 - molto codice da implementare e non sempre di facile gestione
 - difficoltà nella lettura del codice
 - il layout è gestito nel codice.

Layout: ViewGroup

- Gruppi di altri elementi: sia di base che altri gruppi
- Linear Layout: mette in sequenza lineare una serie di oggetti, ha un attributo android:orientation che specifica che tipo di layout è [vertical/orizantal]
- Relative Layout: la posizione degli oggetti è relativa agli altri oggetti, se non si specifica nulla sono messi in alto a sinistra.
- Grid Layout (griglia)
- Frame (contenitore)

Chiameremo root view la radice del file xml, la radice avrà altezza e larghezza = match parent. La posizione della view è specificata dalla distanza da sinistra e dalla distanza dall'alto.

XML attributi

- ID (creazione)
 - android:id="@+id/text"
 - @: il resto della stringa deve essere interpretato
 - +: specifica che stiamo creando (aggiungendo) un nuovo identificatore (id) il cui nome è text
- ID (riferimento)
 - android:id="@id/text" senza il + è un riferimento ad un ID esistente

Nei layout ritornano i padding e margin, hanno lo stesso significato delle controparti css.

Misure pixel: px vs dp

- Screen size: grandezza reale del display (es. 4")
- Screen density: Quanti pixel ci sono nell'unità di area, raggruppati in: low, medium, high e extra high (es 240 dpi = 240 dot-per-inch)
- px= pixel reali (es. 240 dpi x 4" => 960 pixel)
- dp (dip) = density independent pixels, dimensione calcolata su una densità di 160 dpi. Un "dp" ha le dimensioni
- di un "px" a 160 dpi, la dimensione non dipenderà dalla densità reale.

Come standard si usano i dp.

Una buona app dovrebbe fornire alternative per gli oggetti da disegnare (drawable), che crea più risoluzioni della stessa immagine.

Altre unità di misura:

dp, density-independent pixels

sp, scale-independent pixels

– scalato in base alle preferenze dell'utente sulla grandezza del font

pt, points (1/72 di inch)

px, real pixels

mm, millimetri

in, inches

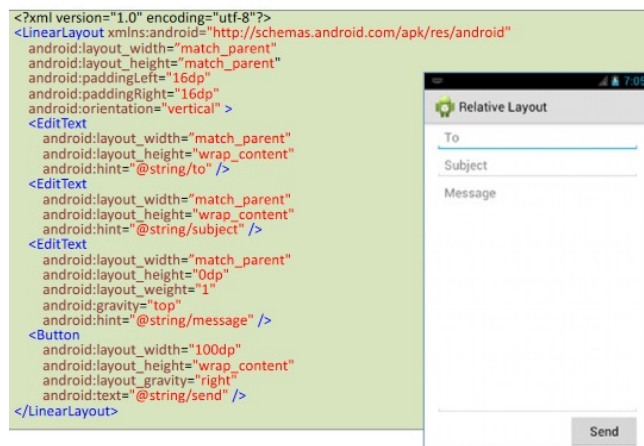
Linear layout

- Posiziona gli elementi uno dopo l'altro (linearmente)
- Orientazione: android:orientation = "vertical" o android:orientation = "horizontal"
- In ogni figlio: android:layout_weight, peso che determina quanto spazio il singolo elemento prende nel Linear Layout.

LL in cui i figli si dividono equamente lo spazio:

- verticalmente : android:layout_height = "0dp" android:layout_weight = "1"
- orizzontalmente: android:layout_width = "0dp" android:layout_weight = "1"

Non viene lasciato spazio vuoto, se lo si vuole si devono inserire degli elementi fittizi. Es: Frame vuoti.

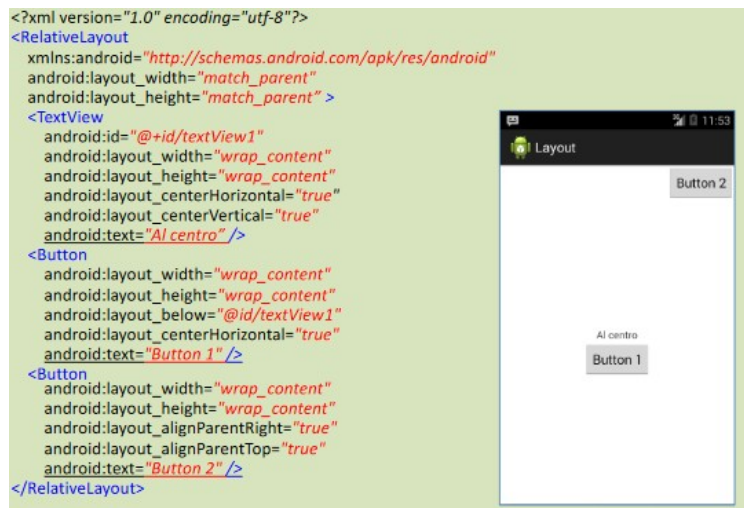


Relativ Layout

La posizione degli elementi è relativa al layout padre e agli altri elementi del layout.

Esempio:

- android:layout_alignParentTop = "true"
- android:layout_centerVertical = "true"
- android:layout_below = "@id/other/object"
- android:layout_toRightOf = "@id/other_object"



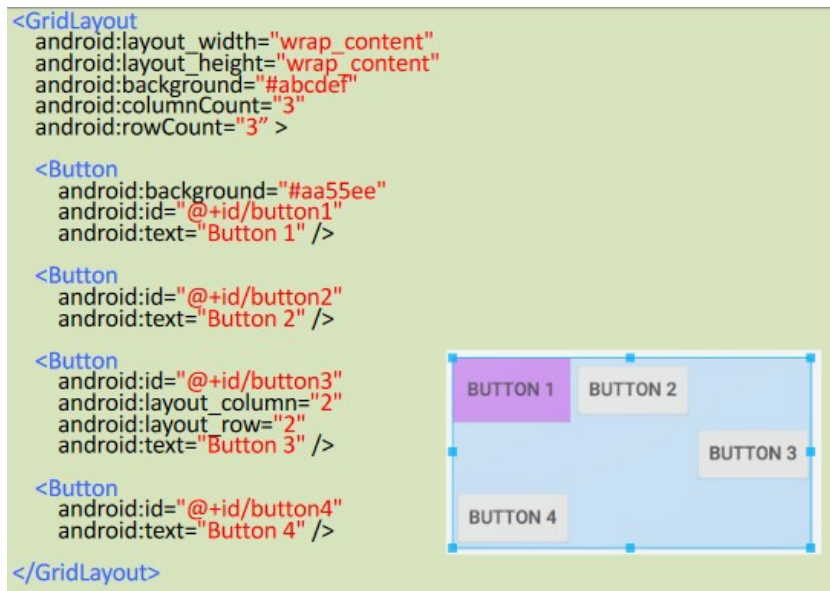
Constraint Layout

Simile al relative permette di specificare la posizione attraverso “vincoli”, comodo quando si lavora con l’editor grafico. Si inseriscono dei “vincoli” che legano la posizione del nuovo oggetto rispetto a quelli esistenti.

Grid View

Visualizza un insieme di elementi il cui numero totale è variabile, è visibile solo una parte e per vederli tutti c’è lo scroll. Nel grid view spesso lo spazio usato è molto più grande di quello a disposizione.

Per questo si usano gli Adapter, che forniscono gli elementi da inserire nel grid view.



List view

Visualizza un insieme di elementi organizzati in una lista, anche qui il numero totale è variabile, visibile solo in parte e c’è lo scroll. E quindi anche qui ci viene in aiuto l’adapter.

Layout widgets

- TextView
- Button
- TextEdit
- ImageView
- CheckBox
- RadioButton
- ecc.

Android studio debugger

Permette di eseguire le app in modalità debug, grazie all'uso dei breakpoint possiamo esaminare il valore delle variabili, ed eseguire passo passo il codice.



• Valuta una espressione

Comandi debugger



• Esecuzione di una singola istruzione



• Entra all'interno di una funzione



• Esci dalla funzione



• Riprendi l'esecuzione fino al prossimo breakpoint

Messaggi di log

sono usati per stampare delle stringhe nel terminale, la sua sintassi è:

Log.x(TAG, "messaggio");

la x può essere:

- d : debug
- e : errore
- i : info
- w : warning
- v : verbose

Widgets

List view

Un widget specifico per le liste divide l'area disponibile in varie sezioni verticalmente, il numero dipende dall'area disponibile e dalla grandezza di ogni elemento, gli elementi sono memorizzati in un array. L'uso di un adapter fornisce gli elementi da visualizzare in base allo scorrimento effettuato.

Lista semplice

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

File xml per il singolo elemento della lista

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <TextView
        android:id="@+id/textViewList"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="" android:padding="10dp"
        android:textSize="22dp"/>
</RelativeLayout>
```

Metodi per creare una ListView

- **Definire un listener per i click sugli elementi**
 - String [] array = {"Pasquale", "Maria", "Michele", "Antonella", "Vincenzo", "Teresa", "Roberto", "Rossella", "Antonio", "Luca", "Liliana", "Stefania", "Francesca", "Andrea", "Marco", "Elisa", "Anna", "Lorenzo"};

- **Definire un adapter**
 - `ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(context, R.layout.list_element, R.id.textViewList, array);`
 - **list_element**: è il file contenente la descrizione xml dell'elemento inserito nell'adapter
 - **textViewList**: è la lista dove inserire gli elementi dell'array adapter.
- **Individuare il widget listview**
 - `listView = (ListView)findViewById(R.id.mylistview);`
- **Associare l'adapter al widget**
 - `listView.setAdapter(arrayAdapter);`

Definire un listener per i click sugli elementi

```
listView.setOnItemClickListener(new OnItemClickListener() {
```

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

Una ListView semplice è una stringa, in un ListView personalizzato ogni elemento ha un proprio layout con dei sotto elementi, però il click è su tutto l'elemento. Con un layout personalizzato con click multiplo si possono cliccare i singoli elementi. Per poter personalizzare gli elementi si deve creare un file di layout e un customAdapter, mentre per il click multiplo dobbiamo creare listeners ad-hoc.

Come creare un adapter customizzato:

```
public class CustomAdapter extends ArrayAdapter<classe dell'oggetto che contiene i dati>
```

```
    ...@Override
    public View getView(int position, View v, ViewGroup parent) {
        if (v == null) {
            Log.d("DEBUG", "Inflating view");
            v = inflater.inflate(R.layout.list_element, null);
        }
        Contatto c = getItem(position);
        Log.d("DEBUG", "contact c="+c);
        Button nameButton;
        Button telButton;
        ImageButton fotoButton;

        nameButton = (Button) v.findViewById(R.id.elem_lista_nome);
        telButton = (Button) v.findViewById(R.id.elem_lista_telefono);
        fotoButton = (ImageButton) v.findViewById(R.id.elem_lista_foto);

        fotoButton.setImageDrawable(c.getPicture());
        nameButton.setText(c.getName());
        telButton.setText(c.getTel());
        fotoButton.setTag(position);
        nameButton.setTag(position);
        telButton.setTag(position);

        return v;
    }
}
```

Nel main:

```
listView = (ListView)findViewById(R.id.mylistview);
customAdapter = new CustomAdapter(this, R.layout.list_element, new ArrayList<Contatto>());
listView.setAdapter(customAdapter);
```

Nell'activity_main.xml:

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

Il nostro file list_element.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <ImageButton
        android:id="@+id/elem_lista_foto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onPictureClick"
        />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

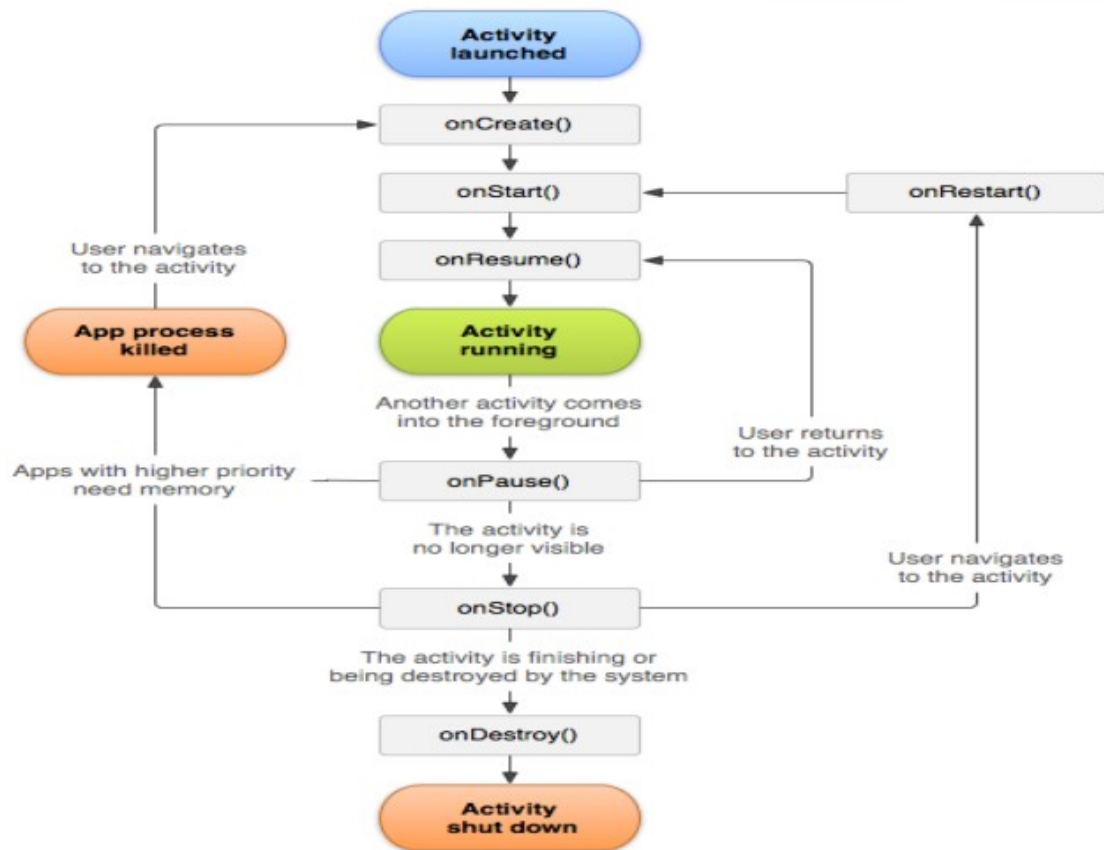
        <Button
            android:id="@+id/elem_lista_nome"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="left|center_vertical"
            android:onClick="onNameClick"
            style="?android:attr/borderlessButtonStyle"
            android:textSize="22dp"/>
        <Button
            android:id="@+id/elem_lista_telefono"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:onClick="onTelClick"
            android:gravity="left"
            style="?android:attr/borderlessButtonStyle"
            android:textSize="12dp"/>

    </LinearLayout>
</LinearLayout>
```

Ciclo di vita delle attività

Ogni Activity ha un proprio ciclo di vita:

- **Attività non esiste**
 1. **onCreate()**
 2. **onStart()**
 3. **onResume()**
- **Attività in esecuzione**
 4. **onPause()**
 5. **onStop()**
 6. **onDestroy()**
- **Attività non esiste**



Metodi nel dettaglio

- **Metodo `onCreate()`:** È necessario implementare questa chiamata di callback, infatti si attiva quando il sistema crea la prima l'attività. Infatti durante la creazione di un'attività si entra nello stato `onCreate`. Questo metodo riceve il parametro `savedInstanceState`, che è un oggetto `Bundle` contenente lo stato precedentemente salvato dell'attività. Se l'attività non è mai esistita prima, il valore dell'oggetto `Bundle` è nullo. In `onCreate` si possono istanziare valori iniziale come `textView` o il contenuto dell'attività.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (savedInstanceState != null) {
        mGameState = savedInstanceState.getString(GAME_STATE_KEY);
    }
    setContentView(R.layout.main_activity);
}
```

- **Metodo `onStart()`:** Quando l'attività entra nello stato "Iniziato", il sistema richiama questa callback. La chiamata `onStart()` rende l'attività visibile all'utente, mentre l'app si prepara all'attivazione dell'attività in primo piano e diventa interattiva. Ad esempio, questo metodo è il punto in cui l'app inizializza il codice che gestisce l'interfaccia utente.

```
public void onStart() {
    super.onStart();
}
```

- **Metodo `onResume()`:** Quando l'attività entra nello stato Resume, arriva in primo piano e quindi il sistema richiama la callback `onResume()`. Questo è lo stato in cui l'app interagisce con l'utente. L'app rimane in questo stato fino a quando non perder il focus. Quando si verifica un evento interruttivo, l'attività entra nello stato di pausa e il sistema richiama la callback `onPause()`.

```
public void onResume() {
    super.onResume();
}
```

- **Metodo onPause():** Il sistema chiama questo metodo quando l'utente lascia l'attività (sebbene non significhi sempre che l'attività viene distrutta); indica che l'attività non è più in primo piano (sebbene possa essere ancora visibile se l'utente è in modalità multi-finestra). Utilizzare il metodo onPause() per mettere in pausa o modificare le operazioni che non dovrebbero continuare (o dovrebbero continuare con moderazione) mentre l'attività è in stato di pausa e che si prevede di riprendere a breve.

```
public void onPause() {
    super.onPause();
}
```
- **Metodo onStop():** Quando l'attività non è più visibile all'utente, entra nello stato Stopped e il sistema richiama la callback onStop(). Ciò potrebbe verificarsi, ad esempio, quando un'attività di nuova apertura copre l'intero schermo. Il sistema può anche chiamare onStop() quando l'attività ha terminato l'esecuzione e sta per essere interrotta. Nel metodo onStop(), l'app dovrebbe rilasciare o regolare le risorse che non sono necessarie mentre l'app non è visibile all'utente.

```
public void onStop() {
    super.onStop();
}
```
- **Metodo onDestroy():** Viene richiamato quando l'attività è distrutta, può essere invocata perché l'attività è terminata o perché il sistema la distrugge temporaneamente per un cambiamento di una configurazione (rotazione schermo).

```
public void onDestroy() {
    super.onDestroy();
}
```
- **Metodo onStart():** è invocato da onStop() nel momento in cui si cerca di ritornare all'attività, infatti è richiamato prima di onStart().

```
public void onStart() {
    super.onStart();
}
```

Quando l'utente lascia l'attività vengono chiamate onPause() e onStop(). Quando ci si ritorna vengono chiamate onStart() e onResume().

Nel momento in cui l'utente ruota il dispositivo l'attività viene prima eliminata quindi sono attivati: onPause(), onStop e onDestroy(); e poi ricreate: onCreate(), onStart() e onResume. Con onDestroy() abbiamo una perdita di stato.

Si può salvare lo stato utilizzando onSaveInstanceState:

```
public void onSaveInstanceState(Bundle savedInstanceState){
    savedInstanceState.putTipoDelDatoDaSalvare("nome chiave", variabile);
    super.onSaveInstanceState(savedInstanceState);
}
```

Lo si può recuperare in onCreate():

```
protected void onCreate(Bundle savedInstanceState){
    if(savedInstanceState != null){
        variabile = savedInstanceState.getTipoVariabileSalvata("nome chiave");
    }
}
```

Backstack

Ogni app normalmente è fatta da più activity e ogni activity ha uno specifico compito un'altra. Un'attività può lanciare un'altra attività oppure lanciare attività che appartengono ad altre app. Quando vengo lanciate più attività vengono inserite in un **backstack**, infatti al lancio di una nuova attività quella corrente è inserita nel backstack, col pulsante indietro l'utente può ritornarci se lo desidera.

Foreground e background

Un task (insieme di attività on cui l'utente interagisce) con le sue attività può essere spostato in background:

- Quando l'utente inizia un nuovo task oppure preme il pulsante Home.

Le attività vengono messe in stato di stop, ma il loro backstack rimane attivo. Nel caso un'attività sia lanciata da più attività si hanno istanze multiple di quest'ultima.

Classe Intent

Serve a lanciare una nuova attività e passare dei dati a quest'ultima. Intent è una descrizione (astratta) di un'operazione da svolgere.

Alcuni dei suoi metodi sono:

- **startActivity**: permette di lanciare una nuova attività.
- **broadcastIntent**: permette di spedire l'intent in broadcast, verrà ricevuto solo dai BroadcastReceiver interessati.
- **startService**: o **bindService**: permette di comunicare con un servizio di background.

Le parti principali di un oggetto intent sono:

- **Action**: cioè l'azione da svolgere.
- **Data**: i dati su cui operare espressi come URI.
 - Si organizzano anche come coppie di (azione, dati), di seguito degli esempi:
 - ACTION_VIEW, content://contacts/people/1
 - ACTION_DIAL, content://contacts/people/1
 - ACTION_DIAL, tel:1112233
- **Category**: informazioni aggiuntive sull'azione da eseguire (es: CATEGORY_BROWSABLE significa che si può usare un browser come Component).
- **Type**: specifica in modo esplicito il tipo (MIME) dei dati. Normalmente il tipo viene dedotto automaticamente.
- **Component**: specifica in modo esplicito l'attività da eseguire (che altrimenti verrebbe dedotta dalle altre informazioni).
- **Extras**: un bundle di informazioni aggiuntive (dati specifici per l'attività).

Risoluzione esplicita e implicita

- Risoluzione esplicita: specificiamo in modo esplicito l'attività (Component) che vogliamo lanciare)
- Risoluzione implicita: La parte component non è specificata, Android sceglie un'attività appropriata in base:
 - Action
 - Type
 - URI
 - Category

Le attività dichiarano le action che possono soddisfarle nel manifesto.

Il metodo startActivityForResult lancia l'attività chiedendo un risultato, REQUEST_CODE serve ad identificare la richiesta.

Il metodo onActivityResult viene richiamato quando si ritorna all'activity di partenza e ci permette di controllare il risultato restituito. Infatti si controllano il REQUEST_CODE, un flag di OK e infine si gestiscono i dati restituiti.

```
@Override
protected void onActivityResult(int request, int result, Intent data) {
    if (request == REQUEST_CODE && result == Activity.RESULT_OK) {
        ...
    }
}
```

Intent Extras

Sono coppie chiave-valore, e permettono di scambiare dati fra activity, i metodi sono:

- putExtra
- getExtra

Intent Flags

Contengono informazioni su come l'intent dovrebbe essere trattato, alcuni dei flag sono:

- FLAG_ACTIVITY_NO_HISTORY: non memorizza l'attività nello backstack.
- FLAG_DEBUG_LOG_RESOLUTION: stampa informazioni aggiuntive quando l'intent viene eseguito, è molto utile in fase di debug se l'intent che vogliamo far eseguire non viene eseguito.

Intent Component

Permette di specificare l'attività "target" da usare quando c'è una sola specifica attività (componente) che deve ricevere l'intent.

```
Intent intent = new Intent(Context context, Class<?> class);  
  
//oppure  
intent.setComponent(...);  
intent.setClass(...);  
intent.setClassName(...);
```

Permessi


Android protegge risorse e dati con un meccanismo di permessi di accesso. Servono a limitare l'accesso a:

- informazioni dell'utente (e.g. i contatti della rubrica)
- servizi con costi (e.g., invio SMS, chiamate tel., accesso a Internet)
- Risorse di sistema (e.g., fotocamera, GPS)
- Vengono rappresentati da stringhe
- Ogni app deve dichiarare nel manifesto i "permessi" che intende utilizzare

es:

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

- L'utente deve accettare i permessi al momento dell'installazione
- I permessi sono divisi in due classi – Normali e "pericolosi"
- I permessi normali vengono concessi senza chiedere nulla all'utente
- I permessi "pericolosi" devono essere approvati dall'utente
 - quando si installa l'app (API < 23)
 - a runtime (API >= 23)

 **Permessi normali (API 23, 6.0)**

Android Mobile Programming – Prof. R. De Prisco

Università di Salerno - Autunno 2018

| | |
|--|--|
| • ACCESS_LOCATION_EXTRA_COMMANDS | • NFC |
| • ACCESS_NETWORK_STATE | • READ_SYNC_SETTINGS |
| • ACCESS_NOTIFICATION_POLICY | • READ_SYNC_STATS |
| • ACCESS_WIFI_STATE | • RECEIVE_BOOT_COMPLETED |
| • BLUETOOTH | • REORDER_TASKS |
| • BLUETOOTH_ADMIN | • REQUEST_INSTALL_PACKAGES |
| • BROADCAST_STICKY | • SET_TIME_ZONE |
| • CHANGE_NETWORK_STATE | • SET_WALLPAPER |
| • CHANGE_WIFI_MULTICAST_STATE | • SET_WALLPAPER_HINTS |
| • CHANGE_WIFI_STATE | • TRANSMIT_IR |
| • DISABLE_KEYGUARD | • USE_FINGERPRINT |
| • EXPAND_STATUS_BAR | • VIBRATE |
| • FLASHLIGHT | • WAKE_LOCK |
| • GET_PACKAGE_SIZE | • WRITE_SYNC_SETTINGS |
| • INTERNET | • SET_ALARM |
| • KILL_BACKGROUND_PROCESSES | • INSTALL_SHORTCUT |
| • MODIFY_AUDIO_SETTINGS | • UNINSTALL_SHORTCUT |

Slide
128

Android Mobile Programming - Prof. R. De Prisco

Slide 129

Università di Salerno - Settembre 2018

Permessi pericolosi (API 23, 6.0)

| Gruppo | Permesso |
|------------|------------------------|
| CALENDAR | READ_CALENDAR |
| | WRITE_CALENDAR |
| CAMERA | CAMERA |
| CONTACTS | READ_CONTACTS |
| | WRITE_CONTACTS |
| | GET_ACCOUNTS |
| LOCATION | ACCESS_FINE_LOCATION |
| | ACCESS_COARSE_LOCATION |
| MICROPHONE | RECORD_AUDIO |
| PHONE | READ_PHONE_STATE |
| | CALL_PHONE |
| | READ_CALL_LOG |
| | WRITE_CALL_LOG |
| | ADD_VOICEMAIL |
| | USE_SIP |
| | PROCESS_OUTGOING_CALLS |

Android Mobile Programming - Prof. R. De Prisco

Slide 130

Università di Salerno - Settembre 2018

Permessi pericolosi (API 23, 6.0)

| Gruppo | Permesso |
|---------|------------------------|
| SENSORS | BODY_SENSORS |
| | SEND_SMS |
| SMS | RECEIVE_SMS |
| | READ_SMS |
| | RECEIVE_WAP_PUSH |
| | RECEIVE_MMS |
| STORAGE | READ_EXTERNAL_STORAGE |
| | WRITE_EXTERNAL_STORAGE |

- Quando l'app richiede un permesso pericoloso
 - se ha già un permesso per lo stesso gruppo viene concesso automaticamente
 - altrimenti viene richiesto all'utente (dialog box) il permesso per il GRUPPO

Threads

- Computazione parallela all'interno di un processo
 - Ogni thread ha il proprio program counter ed il proprio stack
 - Condivide con gli altri thread del processo l'heap e la memoria statica
- Oggetti java.lang.thread
- Implementano l'interfaccia runnable – devono aver il metodo void run()
- Metodi che useremo
 - void start()
 - void sleep(long time)
 - void wait()
 - aspetta che un altro oggetto chiami notify() su questo oggetto
 - void notify()
- Per usare un thread:
 - Creare un oggetto Thread
 - Chiamare il metodo start() del thread
 - che chiamerà il metodo run()
- Android non permette ai thread in background di interagire con l'interfaccia utente
- Solo il main thread può farlo
 - Non possiamo aggiornare l'immagine nel thread creato per caricare l'immagine
- Metodi
 - boolean View.post(Runnable action)
 - void Activity.runOnUiThread(Runnable action)

Async task

Facilitano l'interazione fra background thread e main thread, il thread di background esegue il task e notifica sullo stato di avanzamento, il main thread usa i risultati ottenuti mostrandoli sul display.

Classe Java generica class AsyncTask<Params, Progress, Result> {
 ...
 }

Parametri:

- Params: tipo(di dati) per il lavoro che deve svolgere il background thread
- Progress: tipo(di dati) usato per lo stato di avanzamento
- Result: tipo (di dati) per il risultato del task

AsyncTask.execute()

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

- **onPreExecute ()**, richiamato sul thread dell'interfaccia utente prima dell'esecuzione dell'attività. Questo passaggio viene normalmente utilizzato per impostare un'attività, ad esempio visualizzando una progress bar nell'interfaccia utente.
- **doInBackground (Params ...)**, richiamato sul thread in background subito dopo che **onPreExecute ()** termina l'esecuzione. Questo passaggio viene utilizzato per eseguire calcoli in background che possono richiedere molto tempo. I parametri dell'attività asincrona vengono passati a questo metodo. Il risultato del calcolo deve essere restituito da questo passaggio e verrà passato all'ultima fase. Questo passaggio può anche utilizzare **publishProgress (Progress ...)** per pubblicare una o più unità di avanzamento. Questi valori sono pubblicati sul thread dell'interfaccia utente, nel passaggio a **onProgressUpdate (Avanzamento ...)**.
- **onProgressUpdate (Avanzamento ...)**, richiamato sul thread dell'interfaccia utente dopo una chiamata a **publishProgress (Avanzamento ...)**. I tempi dell'esecuzione non sono definiti. Questo metodo viene utilizzato per visualizzare qualsiasi forma di avanzamento nell'interfaccia utente mentre il calcolo dello sfondo è ancora in esecuzione. Ad esempio, può essere utilizzato per animare una progress bar o mostrare dei log in un campo di testo.
- **onPostExecute (Risultato)**, richiamato sul thread dell'interfaccia utente al termine del calcolo in background. Il risultato del calcolo in background viene passato a questo passo come parametro.

Esempio:

```
class LoadIconTask extends AsyncTask<Integer, Integer, Bitmap> {
    private Integer index = 1;

    @Override
    protected void onPreExecute() {
        progressBar.setVisibility(ProgressBar.VISIBLE);
    }
    @Override
    protected Bitmap doInBackground(Integer... img_ids) {
        // Load bitmap
        Log.d("DEBUG", "index="+index);
        Bitmap tmp = BitmapFactory.decodeResource(getResources(), img_ids[0]);
        /* Simuliamo il ritardo */
        for (int i = 1; i < 11; i++) {
            sleep();
            publishProgress(i * 10);
        }

        return tmp;
    }
    @Override
    protected void onProgressUpdate(Integer... values) {
        progressBar.setProgress(values[0]);
    }
    @Override
    protected void onPostExecute(Bitmap result) {
        progressBar.setVisibility(ProgressBar.INVISIBLE);
        progressBar.setProgress(0);
        imageView.setImageBitmap(result);
    }
    private void sleep() {
        /* Ritardo di 0,5 secondi */
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Frammenti

Un frammento rappresenta una “porzione” dell’UI. Un activity può “ospitare” vari frammenti, quest’ultimi possono essere inseriti e rimossi durante l’esecuzione.

Si possono creare UI con molti frammenti, anche in funzione della grandezza dello schermo.

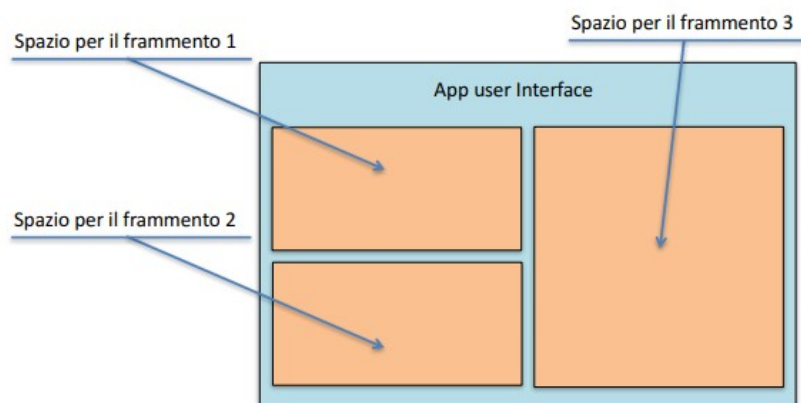
Tipi di frammenti

Ci sono anche alcune sottoclassi che potresti voler estendere, invece della classe Fragment di base:

- **DialogFragment:** Visualizza una finestra di dialogo mobile. L'utilizzo di questa classe per creare una finestra di dialogo è una buona alternativa all'utilizzo dei metodi di helper della finestra di dialogo nella classe activity, poiché è possibile incorporare una finestra di dialogo dei frammenti nel gruppo di frammenti gestiti dall'attività, consentendo all'utente di tornare a un frammento congedato.
- **ListFragment:** Visualizza un elenco di elementi gestiti da un adattatore (come SimpleCursorAdapter), simile a ListActivity. Fornisce diversi metodi per la gestione di una visualizzazione elenco, come ad esempio il callback onItemClick() per gestire gli eventi di clic. (Si noti che il metodo preferito per la visualizzazione di un elenco consiste nell'utilizzare RecyclerView anziché ListView. In questo caso è necessario creare un frammento che includa un RecyclerView nel relativo layout.
- **PreferenceFragmentCompat:** Visualizza una gerarchia di oggetti Preference come un elenco, simile a PreferenceActivity. Questo è utile quando si crea un'attività "Impostazioni" per la propria applicazione.

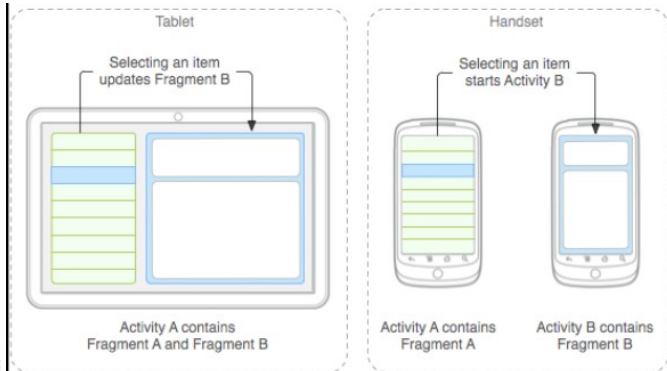
Caratteristiche dei frammenti

- Un frammento è sempre “ospitato” da un’activity
- Un frammento è una sorta di sub-activity
 - ha il suo ciclo di vita
 - che è strettamente legato a quello dell’activity
 - es. se l’activity è in pausa (stato “paused” del ciclo di vita) lo sono anche tutti i suoi frammenti
 - se l’activity è in esecuzione (stato “resumed”) allora i frammenti possono essere gestiti
- La porzione di UI occupata dal frammento deve essere specificata nel layout
 - può essere definita dinamicamente



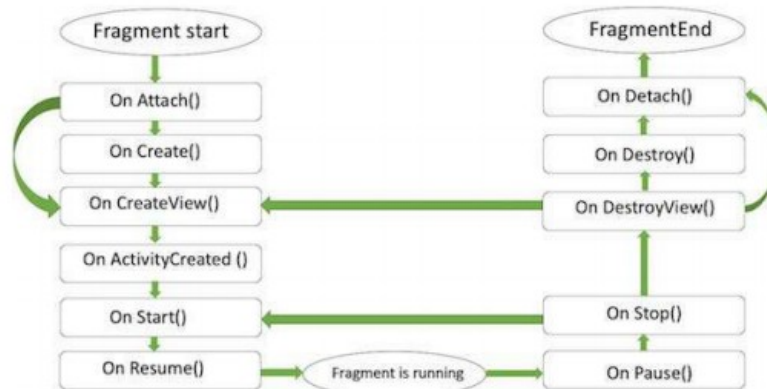
- Filosofia di progettazione
 - Interfacce utente dinamiche
 - in particolare per adattarsi sia a schermi grandi che a schermi piccoli
- Esempio tipico:

- App che gestisce un elenco di elementi
 - es. titoli di articoli di un giornale
- Ogni elemento può essere cliccato per essere esaminato
 - es. visualizzazione dell'articolo
- Si può usare
 - un frammento per l'elenco
 - un frammento per la visualizzazione
- Se lo schermo è piccolo
 - sarà visibile solo uno dei frammenti
 - cliccando un titolo si passerà dal frammento titoli al frammento visualizzazione
- Se lo schermo è grande
 - saranno visualizzati entrambi i frammenti



Creazione dei frammenti

- Istanziare un oggetto Fragment
 - la classe Fragment è simile alla classe Activity
 - proprio ciclo di vita



- Normalmente dovremo implementare almeno
 - onCreate(): inizializzazione come in un activity, NON dobbiamo definire il layout
 - onCreateView(): definiamo il layout. Il metodo deve restituire una View, facciamo l'inflate di un file di layout per il nostro fragment.
 - onPause(): il primo metodo chiamato quando il frammento viene eliminato (si dovrebbero rendere permanenti eventuali cambiamenti altrimenti si perdono)
 - onCreateView è l'equivalente di setContentView nella activity main, view è un oggetto che serve a specificare i parametri di layout.

```

public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup view,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        View v = inflater.inflate(R.layout.example_fragment, container, false);
        return v;
    }
}
  
```

```
public View inflate (int resource, ViewGroup root, boolean attachToRoot)
```

Added in API level 1

Inflate a new view hierarchy from the specified xml resource. Throws `InflateException` if there is an error.

Parameters

| | |
|---------------------|--|
| <i>resource</i> | ID for an XML layout resource to load (e.g., <code>R.layout.main_page</code>) |
| <i>root</i> | Optional view to be the parent of the generated hierarchy (if <i>attachToRoot</i> is true), or else simply an object that provides a set of <code>LayoutParams</code> values for root of the returned hierarchy (if <i>attachToRoot</i> is false.) |
| <i>attachToRoot</i> | Whether the inflated hierarchy should be attached to the root parameter? If false, root is only used to create the correct subclass of <code>LayoutParams</code> for the root view in the XML. |

Returns

The root View of the inflated hierarchy. If root was supplied and *attachToRoot* is true, this is root; otherwise it is the root of the inflated XML file.

Metodo inflate()

Il metodo `inflate()` accetta tre argomenti:

- L'ID della risorsa del layout che si desidera inserire.
- Il `ViewGroup` deve essere il genitore del layout inserito. Passare il contenitore è importante affinché il sistema applichi i parametri di layout alla view radice del layout inserito.
- Un valore booleano che indica se il layout inserito deve essere collegato al `ViewGroup` (il secondo parametro) durante l'inflate. (Nell'esempio sopra riportato, questo è falso perché il sistema sta già inserendo il layout inflate nel contenitore: passando true si creerebbe un gruppo di viste ridondanti nel layout finale.).

Inserimento statico e dinamico dei frammenti

- Un frammento può essere inserito staticamente nel layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
        android:name="com.example.news.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1" android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment>
        android:name="com.example.news.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

- Oppure dinamicamente a runtime:

```
FragmentManager fm= getFragmentManager();
FragmentTransaction ft = fragmentManager.beginTransaction();
ExampleFragment fragment = new ExampleFragment();
ft.add(R.id.fragment_container, fragment);
ft.commit();
```

NOTA: R.id.fragment_container è un ViewGroup nel layout dell'activity che individua la porzione dello schermo da dedicare a questo frammento.

Gestione dei frammenti

Per gestire i frammenti nella nostra activity usiamo il `FragmentManager`:

- `FragmentManager fm= getSupportFragmentManager();` o `getSupportFragmentManager()`

Alcune cose che si possono fare con `FragmentManager` sono:

- Ottenere i frammenti che esistono nell'attività, con `findFragmentById ()` (per i frammenti che forniscono un'interfaccia utente nel layout dell'attività) o `findFragmentByTag ()` (per i frammenti che non forniscono un'interfaccia utente).
- Fare il pop dei Frammenti fuo dallo backstack, con `popBackStack ()` (simulando un comando ritorno indietro dell'utente).
- Registrare un listener per le modifiche allo backstack, con `addOnBackStackChangeListener ()`.

Fragments transaction

È possibile utilizzare `FragmentManager` anche per aprire `FragmentTransaction`, che consente di eseguire transazioni, come aggiungere e rimuovere frammenti:

- `FragmentTransaction ft = fragmentManager.beginTransaction();`

Le operazione effettuabili sono:

- inserire un frammento (*add()*)
- rimuovere un frammento (*remove()*)
- sostituire un frammento (*replace()*)
- applicare la transazione all'activity (*commit()*)

Prima di chiamare `commit ()`, tuttavia, è possibile chiamare `addToBackStack ()`, per aggiungere la transazione a un back stack di transazioni di frammenti. Questo back stack è gestito dall'attività e consente all'utente di tornare allo stato del frammento precedente, premendo il pulsante Indietro.

Spiegazione dettagliata di `addToBackStack()`:

- `addToBackStack()` :per inserire i cambiamenti nel backstack. È usato perchè il backstack considera solo le activity e quindi dobbiamo gestire manualmente i frammenti. Se non chiamiamo `addToBackStack`, quando premiamo back “salteremo” i cambiamenti fatti con i frammenti e non è quello che l'utente si aspetta.

Comunicazione con l'activity

Sebbene un frammento sia implementato come un oggetto indipendente da `FragmentActivity` e possa essere utilizzato all'interno di più attività, una determinata istanza di un frammento è direttamente legata all'attività che lo ospita. In particolare, il frammento può accedere all'istanza `FragmentActivity` con `getActivity ()` e svolgere facilmente attività come trovare una vista nel layout dell'attività:

`View listView = getActivity().findViewById(R.id.list);`

Creazione di eventi callback per l'activity

In alcuni casi, potrebbe essere necessario un frammento per condividere eventi o dati con l'attività e / o gli altri frammenti ospitati dall'attività. Per condividere i dati, si crea un `ViewModel` condiviso. Se è necessario propagare eventi che non possono essere gestiti con un `ViewModel`, è possibile definire un'interfaccia di callback all'interno del frammento e richiedere che la main activity lo implementi. Quando l'attività riceve una chiamata di callback() tramite l'interfaccia, può condividere le informazioni con altri frammenti nel layout, se necessario.

Esempio:

Se un'applicazione di notizie ha due frammenti in un'attività, uno per mostrare un elenco di articoli (frammento A) e un altro per visualizzare un articolo (frammento B), il frammento A deve indicare all'attività quando viene selezionato un elemento dell'elenco che poi dirà di mostrare al frammento B l'articolo collegato. In questo caso, l'interfaccia di `OnArticleSelectedListener` è dichiarata all'interno del frammento A:

```
public static class MyFragment extends Fragment {
    ...
    // Container Activity must implement this interface
    public interface OnArticleSelectedListener {
        public void onArticleSelected(int index);
    }
    ...
}
```

Quindi l'attività che ospita il frammento implementa l'interfaccia `OnArticleSelectedListener` e sovrascrive `onArticleSelected()` per notificare il frammento B dell'evento dal frammento A. Per garantire che la host activity implementi questa interfaccia, il metodo callback `onAttach()` del frammento A (che il sistema chiama quando aggiungendo il frammento all'attività) istanzia un'istanza di `OnArticleSelectedListener` lanciando l'attività passata in `onAttach()`:

```
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        try {
            mListener = (OnArticleSelectedListener) context;
        } catch (ClassCastException e) {
            throw new ClassCastException(context.toString() + " must
implement OnArticleSelectedListener");
        }
    }
    ...
}
```

Il membro `mListener` contiene un riferimento all'implementazione dell'attività di `OnArticleSelectedListener`, in modo che il frammento “A” possa condividere eventi con l'attività chiamando i metodi definiti dall'interfaccia `OnArticleSelectedListener`. Ad esempio, se il frammento A è un'estensione di `ListFragment`, ogni volta che l'utente fa clic su un elemento dell'elenco, il sistema chiama `onListItemClick()` nel frammento, che quindi chiama `onArticleSelected()` per condividere l'evento con l'attività:

```
public static class FragmentA extends ListFragment {

    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        // Append the clicked item's row ID with the content provider Uri
        Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id);
        // Send the event and Uri to the host activity
        mListener.onArticleSelected(noteUri);
    }
    ...
}
```

Il parametro id passato a onItemClick () è l'ID riga dell'elemento selezionato, che l'attività (o altro frammento) utilizza per recuperare l'articolo dal ContentProvider dell'applicazione.

Networking

Per usare i servizi di networking dobbiamo inserire nel file manifest I seguenti permessi:

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

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

Classi che permettono la gestione del networking

- HTTP: org.apache:
 - HttpRequest
 - HttpResponse
- Data formats: JSON, XML
- Classe InetAddress: permette di gestire gli indirizzi IP , ha i seguenti metodi:
 - InetAddress.getByName("www.server.com");
 - InetAddress.getByName("11.22.33.44"): Restituisce l'indirizzo IP:
 - stringa di 32 bit per IPv4
 - stringa di 128 bit per IPv6
- Socket: java.net, crea il canale di comunicazione con il server:
 - Socket(InetAddress addr, int port), socket = new Socket(serverAddr, port);
 - Per leggere e scrivere:
 - getInputStream(socket):
 - BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
in.readLine(), in.read()
 - getOutputStream(socket):
 - PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())), true);
//Autoflush
out.println(strToSend);

Se non si dispone della connessione Internet e/o di un server si può usare l'emulatore. l'emulatore ha come indirizzo locale:10.0.2.2, che sarebbe il localhost:127.0.0.1.

URL e HTTP

Il trasferimento di pagine web è l'operazione più comune, esistono delle classi apposite:

- HttpURLConnection
 - openConnection()
 - getInputStream()
 - e poi si procede come prima leggendo i dati dallo stream.
- classe AndroidHttpClient: stabilisce una connessione HTTP
- classe HttpGet: invia una richiesta GET
- classe responseHandler: legge la risposta

Documenti HTML

I dati contenuti in documenti HTML sono molto difficili da estrarre. Esistono delle librerie che implementano il parsing di documenti HTML, es: JSOUP.

Per utilizzare una libreria abbiamo bisogno dei seguenti passi:

- procurarci il file .jar (es. jsoup-1-1.7.3.jar)
- memorizzarlo nella cartella lib del progetto
- Aggiungere il file jar nella lista delle librerie: Progetto -> Proprietà -> Java Build Path -> Librerie.

Jsoup

La classe Jsoup permette il parsing di documenti HTML e di estrarre singoli parti del documento. Esempi:

Document doc = **Jsoup**.connect("http://en.wikipedia.org/").get();

- **Element** e = doc.getElementById("id");
- **Elements** e = doc.select("[class=id");

Data Storage

- Shared Preferences: dati privati, coppie chiave-valore
- File:
 - File privati dell'app
 - File pubblici (accessibili da altre app)
- Database SQLite: Dati strutturati in database privati

SharedPreferences

Classe SharedPreferences permette di salvare e recuperare dati usando coppie di chiave-valore. Abbiamo 2 metodi della classe Activity:

- getSharedPreferences("filename"): quando si vogliono usare più file di "preferenze" (dati)
- getDefaultSharedPreferences(): quando basta un solo file

Entrambi restituiscono un oggetto SharedPreferences.

NOTA: Attenzione a non usare getPreferences (senza "Shared"), che serve per preferenze non condivise con altre activity dell'app.

- Per usare questo metodo di memorizzazione si istanzia per prima cosa un oggetto di tipo SharedPreferences: SharedPreferences obj;
- Per leggere i dati: si usa "get": Boolean v = obj.getBoolean("KEY");
- Per scrivere i dati:
 - serve un "editor":
 - lo si ottiene con il metodo edit: SharedPreferences.Editor editor = obj.edit();
 - Con l'editor si può usare "put":
 - editor.putBoolean("KEY", bool_value);
 - editor.commit();

FILE

Per ogni app il sistema operativo prevede una directory privata, solo l'app può accedere a questa directory e se l'app viene disinstallata, la directory viene cancellata. Per i file temporanei si può usare una directory cache, però Android cancellerà i file in questa directory SE necessario (quando manca spazio).

Per creare e scrivere un file il procedimento è il seguente:.

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

Esempio:

```
String FILENAME = "hello_file";
String string = "hello world!";
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
    fos.write(string.getBytes());
    fos.close();
```

La modalità può essere:

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

- `MODE_WORLD_READABLE` (leggibile da tutti) (obsoleta da API 17)
- `MODE_WORLD_WRITABLE` (scrivibile da tutti) (obsoleta da API 17)

Per leggere in un file si usa il seguente procedimento:

1. Chiamare `openFileInput(fileName)` che restituisce un `FileInputStream`
2. Leggere dal file (`read()`)
3. Chiudere lo stream (`close()`)

NOTA: È possibile usare un file “statico” mettendolo nella directory “res/raw” dell’applicazione. Lo si può leggere usando `openRawResource` passando come argomento l’identificatore `R.raw`.

Altri metodi:

- `getFilesDir()`: restituisce la directory privata dell’app (dove vengono salvati i file)
- `getDir()`: crea (o apre se esiste) una directory all’interno dello spazio privato dell’app
- `deleteFile()`: cancella il file nello spazio privato
- `fileList()`: restituisce un array di file, quelli presenti nello spazio privato
- `getCacheDir()`:
 - restituisce la directory cache
 - è comunque responsabilità dell’app cancellare i file
 - non si dovrebbe usare la directory cache per file grandi (grandezza massima raccomandata 1MB)

File su External Storage

Android permette l’utilizzo di una memoria esterna per memorizzare i file, tipicamente è una SD card. I file nella memoria esterna sono pubblici (`worldreadable`).

Per usare un external storage occorre richiedere il permesso di lettura/scrittura:

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

L’uso di una memoria esterna porta ad un problema, cioè che può essere rimossa quindi non si può assumere che i file siano sempre disponibili.

NOTA: A partire da Android 4.4, per lo spazio privato non c’è bisogno di permessi.

Esempio codice per vedere se è disponibile una memoria esterna per leggere e scrivere:

```
/* Controlla se è disponibile memoria esterna per leggere e scrivere */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

Esempio codice per vedere se è disponibile una memoria esterna per leggere:

```
/* Controlla se la memoria esterna è disponibile per leggere soltanto*/
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) || Environment.MEDIA_MOUNTED_READ_ONLY.equals(state))
    {
        return true;
    }
    return false;
}
```

Condividere file con altre app

- `getExternalStoragePublicDirectory(type)`
 - `type: DIRECTORY_PICTURES, DIRECTORY_MUSIC , DIRECTORY_RINGTONES, ...`

Esempio: metodo che crea una nuova dir per delle foto nella dir pubblica delle immagini

```
public File getAlbumStorageDir(String albumName) {  
    // Ottieni la directory per la directory delle immagini pubbliche dell'utente.  
    File file = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES), albumName);  
    if (!file.mkdirs()) {  
        Log.e(LOG_TAG, "Directory not created");  
    }  
    return file;  
}
```

Database

Android fornisce un supporto per database SQL all'interno dell'app.

Per usare un database si deve:

- Creare una sottoclasse di `SQLiteOpenHelper` e sovrascrivere il metodo `onCreate()`.
- Quindi si crea un nuovo Helper: `dbHelper = new DatabaseOpenHelper(this);`
- Dal quale si ricava un database: `SQLiteDatabase db = dbHelper.getWritableDatabase();`
- Sul database si possono applicare comandi standard SQL
- Il database (table) viene creato (usando il comando `CREATE`) nel metodo `onCreate()` della sottoclasse `DatabaseOpenHelper`
- Nell'app vengono usati:
 - `db.insert()`
 - `db.delete()`
 - `db.update()`

```
public class MyOpenHelper extends SQLiteOpenHelper {  
    private static final int DATABASE_VERSION = 1;  
    private static final String TABLE_NAME = "dictionary";  
    private static final String CREATE_CMD = "CREATE TABLE " +  
        DICTIONARY_TABLE_NAME + " (" + KEY_WORD + " TEXT, " +  
        KEY_DEFINITION + " TEXT);";  
    MyOpenHelper(Context context) {  
        super(context, TABLE_NAME, null, DATABASE_VERSION);  
    }  
    @Override  
    public void onCreate(SQLiteDatabase db) {  
        db.execSQL(CREATE_CMD);  
    }  
    @Override  
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
        // override necessario }  
    }  
}
```

Cursor: Questa interfaccia fornisce accesso in lettura e scrittura casuale al set di risultati restituito da una query del database.

Esempio:

```
Cursor cursorAll = readAllEntries();  
Cursor cursorAll = db.rawQuery("SELECT * from studenti",null);  
Cursor cursorSelected = readSelectedEntries();  
Cursor cursorSelected = db.rawQuery("SELECT * FROM studenti WHERE nome = ? and voto > ?",  
new String[]{"car%", "25"});  
Cursor cursorSelected = db.rawQuery("SELECT * FROM studenti WHERE nome = Car% and voto > 25",null);
```

```

adapter = new SimpleCursorAdapter(
    getApplicationContext(), //context
    R.layout.list_layout,    //Layout della lista
    cursorAll,               //Il cursore con i dati del database
    DatabaseOpenHelper.columns, // String[] con i nomi delle colonne database
    new int[]{R.id._id, R.id.name, R.id.voto}, //id dei campi nel layout
);
lw.setAdapter(adapter);

```

Grafica

Un'immagine può essere disegnata:

- in un oggetto View: grafica semplice, senza necessità di cambiamenti
- in un oggetto Canvas: grafica complessa, aggiornamenti frequenti

La classe Drawable rappresenta un oggetto che può essere disegnato, questo può essere:

- un'immagine, ma anche un colore, una forma, etc
- ShapeDrawable – una forma
- BitmapDrawable – una matrice di pixels
- ColorDrawable – un colore (uniforme)

L'oggetto Drawable deve essere inserito nell'oggetto View direttamente nel file XML in modo programmatico: View.setImageDrawable().

Animazioni

Android permette di definire delle animazioni da applicare alle immagini. Sono descritte con un file XML:

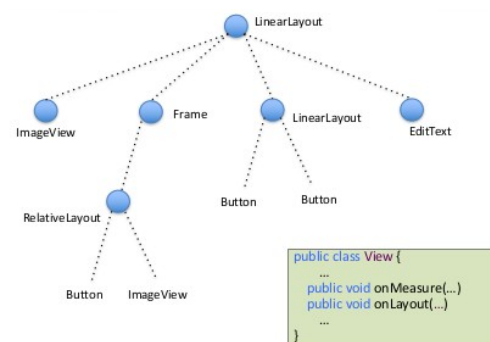
- rotazione
- traslazione
- scaling (dimensione)
- trasparenza
- con controllo di vari parametries: es punto di pivot, velocità, etc.

La classr Animation permette di leggere le animazioni dai file XML e applicarle alle ImageView.

Custom views

Android ha molti widget: Pulsanti, Liste, ImageView, etc. Per esigenze particolare possiamo definire dei widget personalizzati. Permettono un maggiore controllo sulla grafica ovviamente sono più complicati da usare.

Albero delle views



Il meccanismo di layout è diviso in 3 parti:

- Measure
- Layout
- Draw

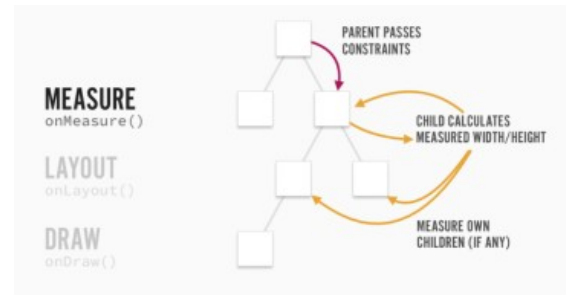


Fase di misurazione

“measured” size (width e height): quanto grande la view vorrebbe essere

size reale: quanto grande la view sarà in realtà

La fase di misurazione ha un’approccio top-down nell’albero, cioè ogni view chiede ai suoi figli quanto vorrebbero essere grandi. La classe è MeasureSpec.



Alcuni metodi di MeasureSpec

```
int widthMode = MeasureSpec.getMode(widthMeasureSpec);
```

```
int width = MeasureSpec.getSize(widthMeasureSpec);
```

```
int heightMode = MeasureSpec.getMode(heightMeasureSpec);
```

```
int height = MeasureSpec.getSize(heightMeasureSpec);
```

width e height hanno valori espressi in pixels, mentre widthMode e heightMode assumono le seguenti costanti:

- MeasureSpec.EXACTLY
- MeasureSpec.AT_MOST
- MeasureSpec.UNSPECIFIED

Ogni view può esprimere la propria preferenza usando le opzioni della classe

ViewGroup.LayoutParams:

- Un numero (di pixel)
- MATCH_PARENT
- WRAP_CONTENT

La misurazione avviene nel metodo onMeasure. Quando il processo di misurazione finisce ogni view deve avere definito:

- measuredWidth
- measuredHeight

In alcuni casi c’è un processo di “negoiazione” fra view parent e view figli, measure() sarà chiamato più volte.

Metodo onMeasure

```
public class MyView extends Views{
    MyView(Context context) {
        super(context);
    }
    ...
    @Override
    public void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

        setMeasuredDimension(getSuggestedMinimumWidth(), getSuggestedMinimumHeight());
    }
    ....
}
```

onMeasure potrebbe essere chiamata varie volte, gli “int” contengono anche dei bit aggiuntivi.

Fase di layout

Visita top-down dell'albero delle view.

View parent decide la grandezza e la posizione (in accordo alle misure fatte nella fase precedente).

Utilizza il metodo `onLayout()`.



Meccanismo di layout

“Container Views” è composto da `RelativeLayout` e

`LinearLayout`. Il meccanismo di layout inizia quando viene chiamato il metodo `requestLayout` su una View dell'albero (solitamente un widget chiama `requestLayout` quando ha bisogno di altro spazio).

A questo punto `requestLayout` mette un evento nella coda degli eventi UI. Quando l'evento viene processato, ogni container view ha la possibilità di interagire con i figli.

Nella fase di Layout le view container comunicano la posizione effettiva ad ogni view figlio.

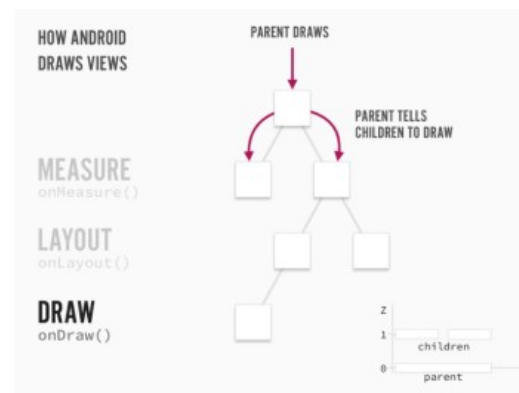
```
public class MyView extends Views{
    ...
    @Override
    public void onLayout (int x1, int y1, int x2, int y2) {
        Log.d("DEBUG","onLayout");
        Log.d("DEBUG","coordinate x1="+x1+" y1="+y1+"
x2="+x2+" y2="+y2);
        int smw = getSuggestedMinimumWidth();
        int smh = getSuggestedMinimumHeight();
        Log.d("DEBUG","onLayout smw="+smw+"
smh="+smh);
        setMeasuredDimension(smw,smh); }
    ....
}
```

Fase di draw

Dopo il posizionamento ogni view viene disegnata usando il metodo `onDraw()`.

Quando c'è un cambiamento:

- `invalidate()`: chiama `onDraw` sulla view
- `requestLayout`: ripete l'intero processo su tutto l'albero.



Il passo più importante nel disegnare una view personalizzata è di sovrascrivere il metodo `onDraw()`. Il parametro su `onDraw()` è un oggetto `Canvas` che la view può utilizzare per disegnare se stesso. La classe `Canvas` definisce i metodi per disegnare testo, linee, bitmap e molte altre primitive grafiche. È possibile utilizzare questi metodi in `onDraw()` per creare l'interfaccia utente personalizzata (UI).

Prima di poter chiamare qualsiasi metodo di disegno, tuttavia, è necessario creare un oggetto `Paint`. Creazione di oggetti paint:

```
private void init() {
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mTextPaint.setColor(mTextColor);
    if (mTextHeight == 0) {
        mTextHeight = mTextPaint.getTextSize();
    } else {
        mTextPaint.setTextSize(mTextHeight);
    }
    mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mPiePaint.setStyle(Paint.Style.FILL);
    mPiePaint.setTextSize(mTextHeight);
    mShadowPaint = new Paint(0);
    mShadowPaint.setColor(0xff101010);
    mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));
    ...
}
```

Sovrascrittura di onDraw():

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Draw the shadow
    canvas.drawOval(
        mShadowBounds,
        mShadowPaint
    );
    // Draw the label text
    canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);
    // Draw the pie slices
    for (int i = 0; i < mData.size(); ++i) {
        Item it = mData.get(i);
        mPiePaint.setShader(it.mShader);
        canvas.drawArc(mBounds,
            360 - it.mEndAngle,
            it.mEndAngle - it.mStartAngle,
            true, mPiePaint);
    }
    // Draw the pointer
    canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
    canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

Multitouch

MotionEvent: rappresenta un movimento registrato da una periferica (penna, trackball, mouse ,dita sul display). Il movimento è rappresentato con:

- ACTION_CODE: cambiamento avvenuto
- ACTION_VALUES: Posizione e proprietà del movimento, tempo, sorgente, pressione e altro

Multitouch display: Permettono il rilevamento di uno o più tocchi. Tramite l'utilizzo di un "Pointer" cioè un il singolo evento (es. un dito che tocca lo schermo).

Un MotionEvent rappresenta infatti un singolo pointer e a volte più di un pointer, in questo caso possiamo accedere ai singoli pointer usando un indice. Ogni pointer ha un ID unico per tutto il tempo in cui esiste. L'indice di un MotionEvent multiplo NON è il pointer ID, il pointer ID è costante mentre l'indice può cambiare per eventi successivi.

MotionEvent ACTION_CODES:

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

Per gestire i MotionEvent:

- `getActionMasked()`: Restituisce l'Action Code dell'evento
- `getPointerCount()`: restituisce il numero di pointer coinvolti
- `getActionIndex()`: indice di un pointer
- `getPointerID(int pointerIndex)`
- `getX(int pointerIndex)`
- `getY(int pointerIndex)`
- `findPointerIndex(int pointerId)`

Android notifica l'oggetto View quando si verifica un evento: `View.onTouchEvent(MotionEvent e)`

- `onTouchEvent()` : deve restituire un Boolean, true, se l'evento è stato consumato, false altrimenti
- Gli oggetti che vogliono ricevere la notifica utilizzano:
 - `View.onTouchListener`
 - `View.setOnTouchListener()`
- `onTouch`: verrà invocata quanto c'è un evento finger down, up o movimento. `onTouch` viene chiamata prima che la View venga notificata dell'evento, anche `onTouch` deve restituire un Boolean: true, se l'evento è stato consumato, false altrimenti

Spesso si ha la necessità di gestire una combinazione di eventi

- Es. Il doppio "click" equivale a
 - `ACTION_DOWN`
 - `ACTION_UP`
 - `ACTION_DOWN`
 - `ACTION_UP`
 - in rapida successione

GestureDetector

Classe `GestureDetector`. permette di riconoscere dei gesti fatti sul display.

Alcuni gesti riconosciuti:

- pressione semplice
- doppia pressione (double "click")
- fling (scorrimento)

NOTA: ADVANCED TOPIC: è possibile anche definire dei gesti personalizzati attraverso un apposito tool di Android e poi riconoscerli tramite il `GestureDetector`

- Bisogna creare un `GestureDetector` che implementa l'interfaccia `GestureDetector.OnGestureListener` interface
- Riscrivere (override) il metodo `onTouchEvent` che viene chiamato in risposta ad un gesto questo metodo delega il riconoscimento del gesto al metodo `GestureDetector.OnGestureListener`

ViewAnimator e ViewFlipper

`View Animator` è una classe per un contenitore di tipo `FrameLayout`, è utilizzata per dare un'animazione di cambiamento fra view.

`Simple ViewAnimator` è una sottoclasse di `ViewAnimator` che:

- crea animazione fra 2 o più view del contenitore
- Solo una view per volta viene visualizzata
- Può anche cambiare views ad intervalli regolari

I metodi sono:

- `showNext()`
- `showPrevious()`

Media player

- AudioManager: controlla le sorgenti audio e l'output (volume)
- MediaPlayer: Play di audio e video
- Sorgente dati :
 - Risorse locali
 - URI (interni)

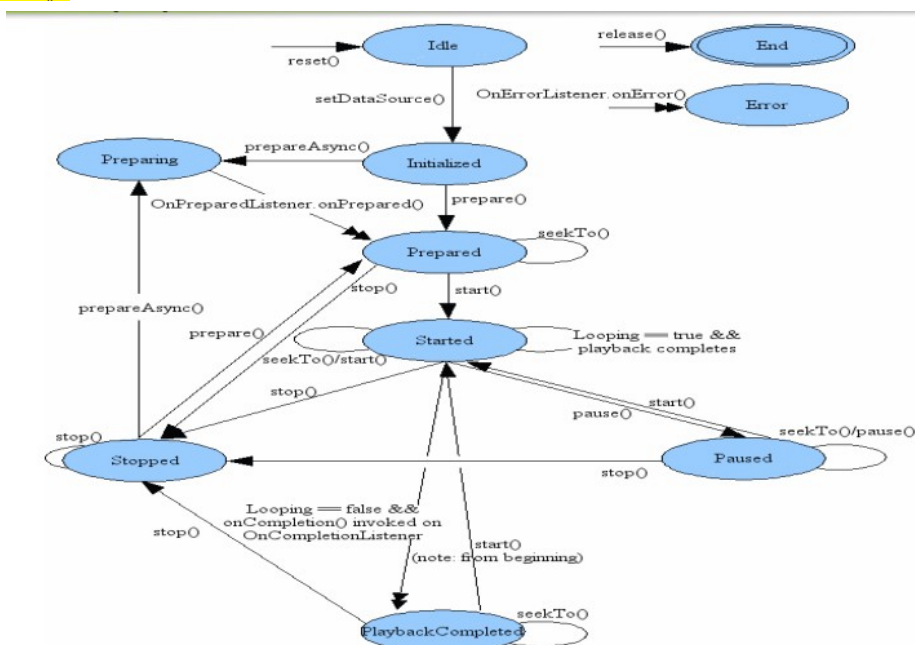
URL

Play di un file in res/raw:

```
MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file); mediaPlayer.start(); //  
no need to call prepare(); create() does that for you
```

Play di un file da URL

```
String url = "http://....."; // your URL here  
MediaPlayer mediaPlayer = new MediaPlayer();  
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);  
mediaPlayer.setDataSource(url);  
mediaPlayer.prepare(); // might take long! (for buffering, etc)  
mediaPlayer.start();
```



Rilasciare la risorsa

```
mediaPlayer.release();  
mediaPlayer = null;
```

Metodi

- `mediaPlayer.start();`
- `mediaPlayer.pause();`
- `mediaPlayer.stop();`
- Attenzione all'uso asincrono (`prepareAsync`)
 - necessario per non appesantire l'app
 - Sensoririchiede più attenzione
 - Play da fare in/dopo `onPrepareListener.onPrepared()`

- Audio focus. Poichè c'è un solo canale di output, l'utilizzo da parte di più applicazioni può essere un problema, es. se stiamo ascoltando musica potremmo non sentire l'arrivo di un messaggio. È possibile gestire l'accesso contemporaneo usando l'audio focus
 - un'app richiede l'audio focus per usare l'audio
 - se lo perde deve o smettere di suonare o abbassare il proprio volume

Sensori

Alcuni sensori richiedono il permesso nell'manifest.xml.

- Molti smartphones, tablet hanno sensori
 - di movimento
 - forze di accelerazione e di rotazione
 - accelerometri, bussola, giroscopio
 - di ambiente
 - temperatura, pressione, umidità
 - termometri, barometri
 - di posizione
 - posizione fisica
 - magnetometro, bussola, giroscopio
- Forniscono dati "grezzi", l'accuratezza dipende dalla qualità
- SensorManager ci dice i sensori disponibili e le caratteristiche del singolo sensore:
 - range massimo
 - accuratezza
 - etc.
- ci permette di:
 - leggere i dati grezzi del sensore
 - usare Listeners sui cambiamenti dei dati
- Pochi device hanno tutti i tipi di sensori a volte più di un sensore dello stesso tipo
 - ecco un elenco parziale:
- L'attività deve implementare SensorEventListener

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

- Poi si deve controllare se il sensore esiste:

| Sensore | Tipo | Descrizione |
|--------------------------|---------|--|
| TYPE_ACCELEROMETER | Hw | Misura le forze in m/s ² applicate alla device (inclusa la gravità) nelle 3 direzioni (x,y,z) |
| TYPE_AMBIENT_TEMPERATURE | Hw | Misura la temperatura dell'ambiente in gradi centigradi |
| TYPE_GRAVITY | Hw o Sw | Misura le forze di gravità sui 3 assi (x,y,z) |
| TYPE_GYROSCOPE | Hw | Misura la velocità di rotazione in rad/s nelle 3 direzioni (x,y,z) |
| TYPE_MAGNETIC_FIELD | Hw | Misura il campo magnetico sui tre assi |
| TYPE_ORIENTATION | Sw | Misura la rotazione riferita ai 3 assi |
| TYPE_RELATIVE_HUMIDITY | Hw | Misura la % di umidità dell'ambiente |
| TYPE_LIGHT | Hw | Misura la luminosità dell'ambiente |

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT) != null){
// Success! There's an ambient light sensor.
} else {
// Failure! No light sensor
}
```

```

@Override
protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(this, mLight,
        SensorManager.SENSOR_DELAY_NORMAL); }

```

```

@Override
protected void onPause() {
    super.onPause();
    mSensorManager.unregisterListener(this);
}

```

- Velocità di campionamento:
 - SENSOR_DELAY_NORMAL (0,2sec)
 - SENSOR_DELAY_GAME (0,02sec)
 - SENSOR_DELAY_UI (0,06sec)
 - SENSOR_FASTEST (0sec)
- Registrazione e rilascio in onResume e onPause per evitare di consumare la batteria.

```

@Override
public final void onCreate(Bundle savedInstanceState) {
    ....
    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
}

```

```

@Override
public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Do something here if sensor accuracy changes.
}

```

```

@Override
public final void onSensorChanged(SensorEvent event) {
    // The light sensor returns a single value, Many sensors return 3 values, one for each axis.
    float lux = event.values[0];
    ...
}

```

Toast, Dialog e Notifiche

Toast:

```

public void showToast(View v) {
    Toast.makeText(getApplicationContext(), "Toast!", Toast.LENGTH_LONG).show();
}

```

Toast personalizzato:

```

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

```

Dialog:

```

public void showDialog(View v) {
    DialogInterface.OnClickListener dialogClickListener = new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            switch (which) {
                case DialogInterface.BUTTON_POSITIVE:
                    Toast.makeText(getApplicationContext(), "Ok!", Toast.LENGTH_LONG).show();
                    break;
                case DialogInterface.BUTTON_NEGATIVE:
                    Toast.makeText(getApplicationContext(), "Azione annullata", Toast.LENGTH_LONG).show();

```

```

        break;
    }
}
};
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Stai per ripartire da capo. Sei sicuro?")
    .setPositiveButton("Sì", dialogClickListener)
    .setNegativeButton("No", dialogClickListener).show();

return;
}

```

Mostra notifica:

```

public void showNotification(View v) {
    Notification.Builder notificationBuilder = new Notification.Builder(
        getApplicationContext())
        .setTicker("Messaggio breve")
        .setSmallIcon(R.drawable.pacman_ghost)
        .setAutoCancel(true)
        .setContentTitle("Titolo notifica")
        .setContentText("Testo della notifica");
    // Pass the Notification to the NotificationManager:
    NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    mNotificationManager.notify(1,notificationBuilder.build());
}

```

Cancella notifica:

```

public void cancelNotification(View v) {
    NotificationManager mNotificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    mNotificationManager.cancel(1);
}

```

Alarms

Permettono di eseguire intent in funzione di specifici eventi, gli intent sono lanciati in base al tempo.

Un'applicazione che usa un alarm riesce ad eseguire porzioni di codice anche se l'applicazione è terminata. Un alarm è attivo anche se il telefono va in modalità di sleep

- l'alarm può causare la ripresa dell'attività
- oppure potrà essere gestito quando l'utente rimette il telefono in modalità normale

Gli alarms rimangono attivi fino a quando

- vengono cancellati
- la periferica viene spenta

Esempi di alarms:

- app per gli MMS: usa alarm per controllare periodicamente i messaggi non spediti (retry scheduler)
- Settings: usa un alarm per rendere la periferica non visibile via Bluetooth dopo un determinato tempo

Per usare gli alarm in un'app: ottenere un riferimento ad AlarmManager con
getSystemService(Context.ALARM_SERVICE)

Creare alarms

- void set(int type, long triggerAtTime, PendingIntent i)
- void setRepeating(...), setInexactRepeating(...)

A partire dall'API level 19 (KitKat) gli alarm non sono "esatti": il SO operativo può modificare i triggerTime per minimizzare wakeups e l'uso della batteria

- ELAPSED_REALTIME
 - lancia il "pending" intent dopo un determinato tempo (ma non fa il wakeup della device)
- ELAPSED_REALTIME_WAKEUP

- Se la device non è attiva fa il wakeup e lancia il “pending” intent
- RTC
 - lancia il pending event ad un determinato orario
- RTC_WAKEUP
 - come prima ma fa il wakeup se serve

Content Providers, Broadcast, Services

- 4 componenti fondamentali di Android
 - Activity
 - Broadcasts
 - Content Providers
 - Services
- Finora abbiamo parlato delle activity
 - servono per lo sviluppo delle app!
 - Le altre componenti sono di ausilio e servono in casi particolari, ma in alcuni casi sono estremamente utili

Broadcast

Le app Android possono inviare o ricevere messaggi broadcast dal sistema Android e da altre app Android, in modo simile al modello di progettazione publish-subscribe. Queste trasmissioni vengono inviate quando si verifica un evento di interesse.

Ad esempio, il sistema Android invia trasmissioni quando si verificano vari eventi di sistema, ad esempio all'avvio del sistema o all'avvio della ricarica del dispositivo. Le app possono anche inviare trasmissioni personalizzate, ad esempio per notificare ad altre app qualcosa di cui potrebbero essere interessati (ad esempio, alcuni nuovi dati sono stati scaricati).

Le app possono registrarsi per ricevere trasmissioni specifiche. Quando viene inviata una trasmissione, il sistema inoltra automaticamente le trasmissioni alle app che si sono abbonate per ricevere quel particolare tipo di trasmissione.

In generale, le trasmissioni possono essere utilizzate come un sistema di messaggistica attraverso le app e al di fuori del normale flusso di utenti.

Content Providers

I fornitori di contenuti gestiscono l'accesso a un insieme strutturato di dati. Incapsulano i dati e forniscono meccanismi per la definizione della sicurezza dei dati. I fornitori di contenuti sono l'interfaccia standard che collega i dati in un processo con codice in esecuzione in un altro processo. Quando si desidera accedere ai dati in un provider di contenuti, si utilizza l'oggetto ContentResolver nel contesto dell'applicazione per comunicare con il provider come client. L'oggetto ContentResolver comunica con l'oggetto provider, un'istanza di una classe che implementa ContentProvider. L'oggetto provider riceve richieste di dati dai client, esegue l'azione richiesta e restituisce i risultati.

Non è necessario sviluppare il proprio provider se non si intende condividere i dati con altre applicazioni.

Tuttavia, è necessario il proprio provider per fornire suggerimenti di ricerca personalizzati nella propria applicazione. È inoltre necessario il proprio provider se si desidera copiare e incollare dati o file complessi dall'applicazione ad altre applicazioni.

Lo stesso Android include fornitori di contenuti che gestiscono dati come audio, video, immagini e informazioni di contatto personali. Puoi vedere alcuni di essi elencati nella documentazione di riferimento per il pacchetto android.provider. Con alcune restrizioni, questi provider sono accessibili a qualsiasi applicazione Android.

Services

Un servizio è un componente dell'applicazione che può eseguire operazioni di lunga durata in background e non fornisce un'interfaccia utente. Un altro componente dell'applicazione può avviare un servizio e continua a funzionare in background anche se l'utente passa a un'altra applicazione. Inoltre, un componente può collegarsi a un servizio per interagire con esso e persino eseguire la comunicazione interprocesso (IPC). Ad esempio, un servizio può gestire transazioni di rete, riprodurre musica, eseguire I / O file o interagire con un fornitore di contenuti, tutto in background.

Broadcast ed eventi

- Parleremo:
 - della classe BroadcastReceiver
 - di come un “ricevitore di broadcast” deve essere “registrato”
 - dei modi in cui gli eventi possono essere inviati ai ricevitori di broadcast
 - di come i ricevitori ricevono una notifica di un evento
 - di come i ricevitori gestiscono la notifica
- serve per ricevere e “reagire” ad eventi
 - eventi sono rappresentati da Intent
- un “ricevitore” deve “registrarsi” dichiarando gli eventi ai quali è interessato
 - es. esiste un BroadcastReceiver che ha il compito di spedire messaggi MMS
- Quando un'altra componente crea un MMS invia un evento (Intent)
 - l'Intent viene mandato in broadcast al sistema
 - Il ricevitore lo intercetta e spedisce il messaggio
- Il ricevitore riceve l'Intent tramite il metodo
 - onReceive(Context c, Intent i)

Riassumendo:

1. Il “ricevitore” si “registra” usando registerReceiver() (disponibile nel LocalBroadcastManager o nel Context)
2. L'evento viene creato (da qualche altra componente del sistema)
3. Android notifica il ricevitore chiamando onReceive()

Lo si può fare

- staticamente nel Manifesto dell'app
- dinamicamente usando registerReceiver()

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
<application
    ...
    <receiver
        android:name=".My_Receiver"
        android:exported="false"
        <intent-filter>
            <action android:name="it.unisa.mp.MY_ACTION" />
        </intent-filter>
    </receiver>
</application>
</manifest>
```

Registrazione BroadcastReceiver

Se la registrazione è statica il ricevitore viene registrato durante il Boot del sistema (oppure quando l'app viene installata). Se la registrazione è dinamica il ricevitore viene registrato quando si chiama:

- LocalBroadcastManager.registerReceiver()
 - per i broadcast locali all'app

- Context.registerReceiver()
 - per i broadcast system-wide
- è possibile anche revocare la registrazione usando unregisterReceiver()

Spedizione broadcast

Per spedire un messaggio si usa il metodo

- sendBroadcast(Intent i)
- sendBroadcast(Intent i, String permission)

Se si specifica anche una stringa di permesso l'intent verrà consegnato solo ai ricevitori che hanno il permesso: il permesso lo deve avere l'app nel Manifesto

sendBroadcast(Intent i) è disponibile sia nel LocalBroadcastManager che nel Context. Chiaramente si utilizza il primo per messaggi locali all'app ed il secondo per messaggi system-wide. sendBroadcast(Intent i, String permission) è disponibile solo nel Context.

Esempi di altri eventi globali.:

- android.intent.action.AIRPLANE_MODE
- android.intent.action.BATTERY_LOW
- android.intent.action.DATA_SMS_RECEIVED
- android.intent.action.DATE_CHANGED
- android.intent.action.DEVICE_STORAGE_LOW
- android.intent.action.TIMEZONE_CHANGED
- android.intent.action.TIME_TICK
- android.intent.action.USER_PRESENT
- android.intent.action.WALLPAPER_CHANGED

ContentProvider

Rappresentano Contenitori Dati progettati per condividere le informazioni fra le applicazioni. Per accedere ad un ContentProvider si utilizza un ContentResolver

- interfaccia simile a quella di un database
- comandi SQL-like: QUERY, INSERT, UPDATE, DELETE, etc

in più, notifiche su cambiamenti dei dati.

Per usare il resolver occorre recuperare un suo riferimento chiamando:

Context.getContentResolver()

ContentProviders standard:

- Browser (info su bookmarks, history)
- Call Log (info sulle chiamate)
- Contact (info sui contatti presenti in rubrica)
- Media (lista dei file multimediali utilizzabili)
- UserDictionary (lista delle parole digitate)
- ... molti altri

I dati contenuti in un provider sono memorizzati in tabelle:

- Gli utenti possono far riferimento ad uno specifico ContentProvider usando un URI
- URI: content://authority/path/id
 - authority: specifica il content provider
 - path: specifica la tabella
 - id: specifica un particolare record
- Esempi di URI
 - content://com.android.contacts/contacts/
 - Authority è com.android.contacts

- La tabella richiesta è “contacts”
- Non c’è nessun ID, quindi l’URI identifica l’intera tabella dei contatti

Per ottenere i dati usiamo una query ed un Cursor

- ContentResolver.query():
 Cursor query(Uri uri,
 String[] projection \\
 String selection \\
 String[] args \\
 String sortOrder) \\
 ordinamento

Restituisce un Cursor che ci permette di iterare sull’insieme di record restituiti dalla query

Services

Servono a eseguire operazioni complesse che possono richiedere molto tempo: es. scaricare un file d Internet, sincronizzare informazioni locali con un server

Vedremo:

- La classe Services
- come usare dei Services esistenti
- come definire dei nuovi Services

Services non interagiscono con l’utente non c’è una UI. Servono per eseguire delle operazioni in background. L’app interagisce con il servizio. La prima cosa da fare è far partire il Service: Context.startService(Intent i).

Una volta partito, il Service può continuare la sua esecuzione fino a che il device è acceso

- potrebbe anche essere interrotto se occorrono le risorse che esso usa
- potrebbe anche terminare volontariamente

Nell’utilizzo tipico un Service fatto partire da un’app termina la propria esecuzione dopo aver eseguito l’operazione richiesta, per default, il Service gira nel main thread dell’app che lo ha fatto partire, in alcuni casi deve essere esplicitamente fatto girare su un thread separato.

Le componenti che vogliono interagire con un Service devono effettuare un “bind”:

Context.bindService(Intent service, ServiceConnection conn, int flags);

Il binding permette di:

- inviare richieste
- ricevere risposte

Se al momento delle richiesta di bind il Service non è ancora attivo

- viene fatto partire
- rimane attivo fino a quando c’è almeno un client connesso

Occorre dichiarare il Service nel Manifesto:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <application
        ...
        <service android:name="MyService" />
    </application>
</manifest>
```



Fine del corso!