Lezione 13

Programmazione Android

- Storage temporaneo
  - **Salvataggio temporaneo dello stato**
- Storage permanente
  - **Preferenze**
    - Shared & Private Preferences
    - PreferenceScreen e PreferenceActivity
  - **Accesso al File System**
  - **Accesso a Database**
- Condivisione di dati
  - **Content Provider**

Accesso a Database

SQLite

- Android incorpora una versione di **SQLite**
  - Database di uso generale
  - Particolarmente “piccolo”, implementato come una libreria dinamica (.so)
  - Non adatto a grandi quantità di dati, ma efficiente per piccoli database
- Ogni applicazione ha un insieme di database SQLite associato
  - Solo l'app può accedere ai “suoi” database
  - Si posso esporre i dati ad altri tramite **Content Provider**

SQLiteDatabase

- 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”
- Vedremo brevemente entrambi

SQLiteDatabase

- Per aprire o creare un database si possono usare vari metodi statici di SQLiteDatabase
  - Molte varianti overloaded
  - **openDatabase(*path*, *factory*, *flags*)**
    - *path* è il pathname del DB
    - *factory* è la classe da invocare per creare i **Cursor**
      - La cosa “normale” è passare **null** e usare la factory di default
    - *flags* indica il modo di apertura, bitmask fra:
      - OPEN_READWRITE
      - OPEN_READONLY
      - CREATE_IF_NECESSARY
      - NO_LOCALIZED_COLLATORS

Restituisce un'istanza di SQLiteDatabase

SQLiteDatabase

- Un gergo usato frequentemente è
  - SQLiteDatabase db =
    SQLiteDatabase.openOrCreateDatabase(*path*,null);
- Più raramente, si usa creare un **database in memoria** (non salvato in un file!) per memorizzare in maniera temporanea dati su cui sia utile operare in maniera relazionale
  - SQLiteDatabase db =
    SQLiteDatabase.create(null);

Un DB in memoria è molto veloce, ma viene cancellato al momento della close()!

Altre operazioni sul DB

- Il Context (e quindi, anche l'Activity) offre alcune altre funzioni di utilità
- Si tratta di funzioni di gestione “globale” del DB
  - String [] **databaseList()** – restituisce l'elenco dei nomi di DB associati al contesto
  - boolean **deleteDatabase(*nome*)** – cancella un DB
  - String **getDatabasePath(*nome*)** – restituisce il path assoluto di un DB
  - SQLiteDatabase **openOrCreateDatabase(*nome*, *modo*, *factory*)** – apre o crea un DB

Eseguire istruzioni SQL

- 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

Eseguire istruzioni SQL

- Nel caso si usino dei *placeholder* nella query SQL, occorre usare una versione di **execSQL()** che prende anche gli argomenti
  - s=”INSERT INTO Aule (nome, edificio) VALUES (?,?)”;
  - Object[] a = { “A”, “Marzotto B” };
  - db.**execSQL(s,a)**;
- La cosa può anche essere spezzata in più passi
  - SQLiteStatement st=db.**compileStatement(s)**;
  - st.**bindString(1, “A”)**; st.**bindString(2,”Marzotto B”)**;
  - st.**execute()**;

SQL a programma

- 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*)**

SQL a programma

- Il costo è (affatto)
- SQLite anziché specific
  - **delete(*tabella*, *where*, *args*)**
  - **insert(*tabella*, *nullcolumn*, *valori*)**
  - **replace(*tabella*, *nullcolumn*, *valori*)**
  - **update(*tabella*, *valori*, *where*, *args*)**

Esempi

where = “edificio=?”

args = new String[] {“Marzotto D”}

valori è un **ContentValues** – l'ennesima mappa chiavi-valori (fornisce una serie di metodi put() e getAs*Tipo*() e simili). La chiave è il nome della colonna nella tabella.

nullcolumn è il nome di una colonna in cui inserire un valore NULL, usato solo se valori è la mappa vuota (altrimenti, può essere **null**)

Esempio – INSERT

```
ContentValues cv = new ContentValues();

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

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

- 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
  - Esiste anche una versione che accetta byte[]

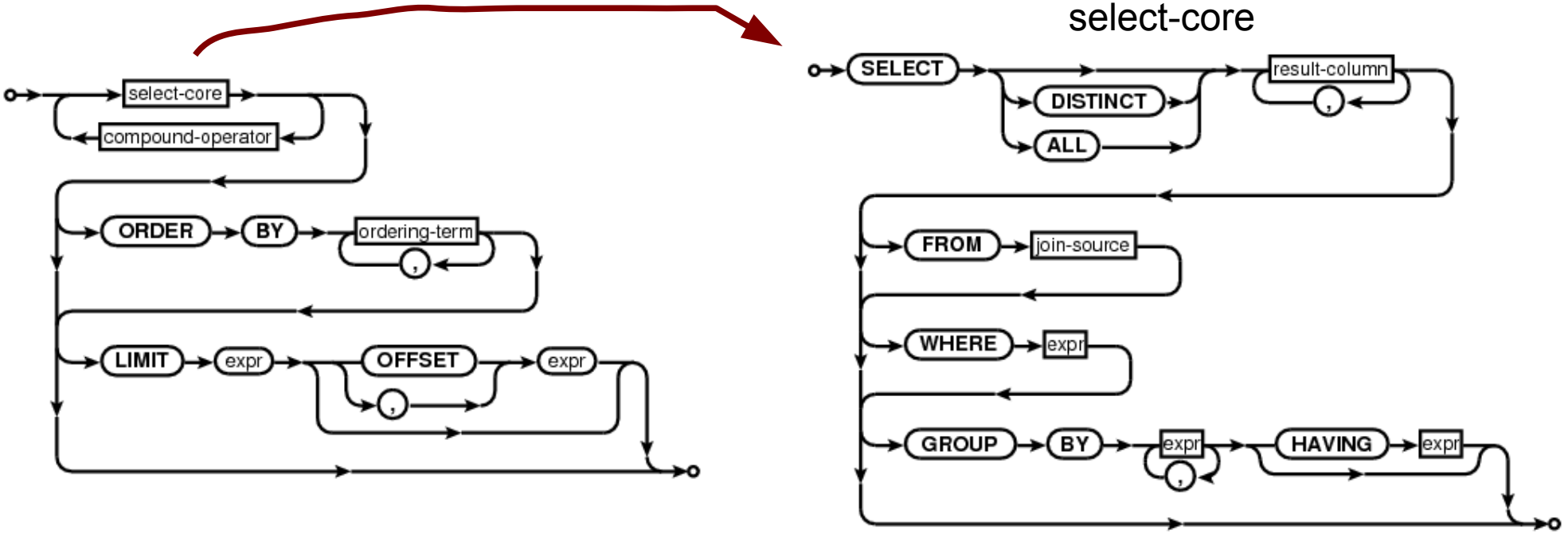
Esempio – UPDATE

```
ContentValues cv = new ContentValues();

cv.put("edificio", "Fibonacci C");

db.update("Aule", cv, "edificio=?", new String[] {"Marzotto C"});
```

- Ovviamente, sarebbe possibile...
  - inserire più coppie nel ContentValue
    - e aggiornare diversi campi insieme)
  - usare condizioni WHERE più complesse
    - Con o senza argomenti

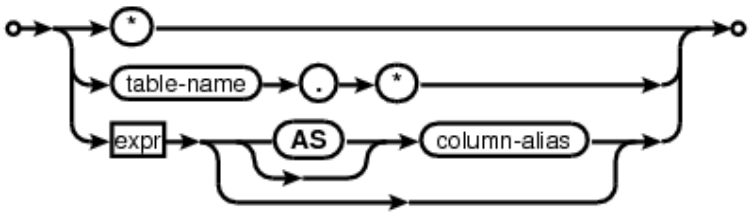
Eseguire una SELECT

- L'operazione più frequente su un DB è normalmente la SELECT
- È anche il comando più complesso!



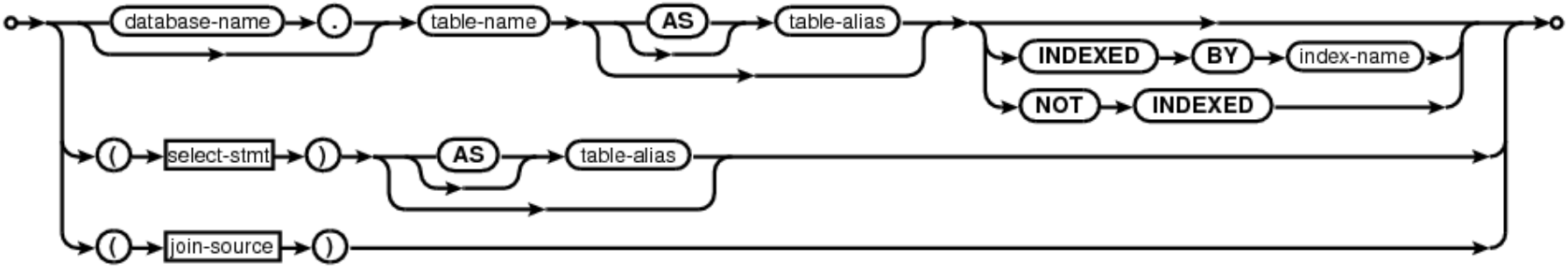
Eseguire una SELECT

result-column

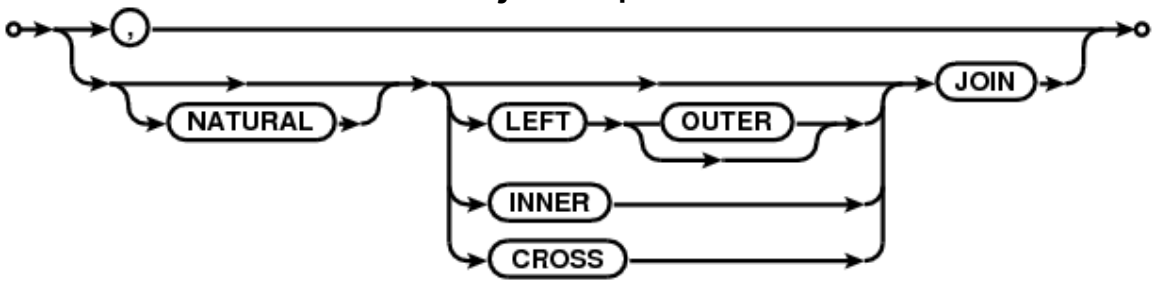


join-source

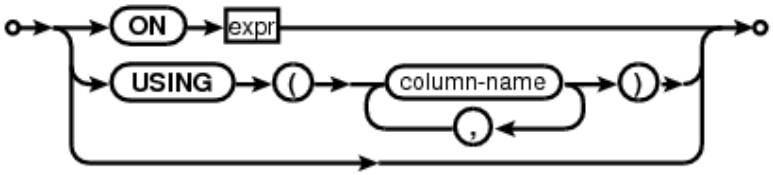
single-source



join-op



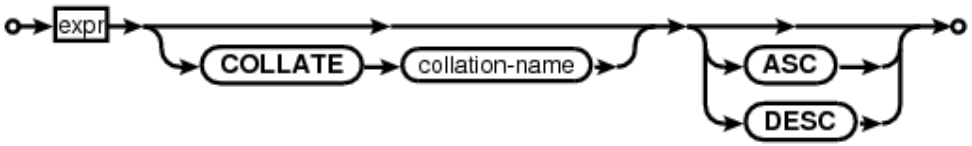
join-constraint



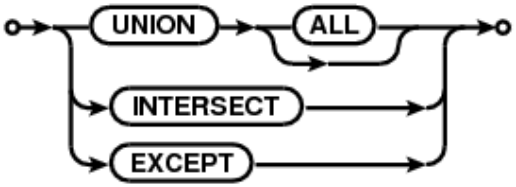
Grafici di sintassi da www.sqlite.org

Eseguire una SELECT

ordering-term



compound-operator



- I casi in cui si usa l'intera potenza espressiva di SELECT sono **rarissimi** (mai visto uno)
- Si usano le clausole più comuni:
  - SELECT *colonne* FROM *tabelle* WHERE *condizione*
  - DISTINCT, GROUP BY, ORDER BY, HAVING, LIMIT

Eseguire una SELECT

- Anche in questo caso, abbiamo due possibilità
  - Eseguire la SELECT come statement SQL
    - Cursor **rawQuery(*sql*, *args*)**
  - Eseguire la SELECT a programma
    - Cursor **query(*distinct*, *tabella*, *colonne*, *selezione*, *args*, *groupby*, *having*, *orderby*, *limit*)**
    - Esistono alcune varianti overloaded che hanno un sottoinsieme degli argomenti
    - La maggior parte dei parametri può essere **null**
- Il **Cursor** ci consente di scorrere i risultati

Esempi – SELECT

```
String sql=”SELECT * FROM Aule WHERE edificio='Marzotto B'”;

Cursor cur = db.rawQuery(sql, null);
```

```
Cursor cur = db.query( false,
                       “Aule”,
                       new String[]{“nome”, “edificio”},
                       “edificio=?”,
                       new String[]{“Marzotto B”},
                       null, null,  null
                );
```

Il Cursor

- 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
- Offre metodi per:
  - Spostamento nella tabella
  - Controllo di condizioni
  - Accesso ai valori dei campi

Il Cursor – posizionamento

- Appena ottenuto, un Cursor è posizionato sul primo record (se c'è!) dei risultati
- Metodi di spostamento
  - move(offset), moveToPosition(indice)
  - moveToFirst(), moveToLast(), moveToNext(), moveToPrevious()
- Metodi per leggere la posizione
  - getPosition()
  - IsFirst(), isLast(), isBeforeFirst(), isAfterLast()

Il Cursor – informazioni

- I metodi di controllo consentono di ottenere informazioni sul cursore stesso e sulla tabella
  - isClosed() – il cursore è stato chiuso, fine dei giochi
  - getColumnCount() – quante colonne ha la tabella
  - getColumnNames() – nomi delle colonne
  - getCount() – quante righe ha la tabella
- Alcuni di questi metodi (es., getCount()) possono essere costosi!

Il Cursor – lettura campi

- Il Cursor offre una serie di metodi **get*Tipo*(*i*)**
  - Il *Tipo* è il tipo base Java corrispondente al tipo SQL del campo
  - *i* è l'indice numerico della colonna che vogliamo leggere
  - Il risultato è il valore dell'*i*-esimo campo del record puntato dal cursore
- Il metodo **getType(*i*)** restituisce (una codifica de) il tipo dell'i-esima colonna
  - FIELD_TYPE_NULL, FIELD_TYPE_INTEGER, FIELD_TYPE_FLOAT, FIELD_TYPE_STRING, FIELD_TYPE_BLOB

Aggiornare una query

- Può accadere che, dopo aver fatto una query, si voglia (o si debba) sospendere l'elaborazione
- **deactivate()** "disattiva" il cursore
  - Ogni tentativo di usare un cursore disattivato da errore

Deprecato
(da api level 16)

- **requery()** ripete la query originale di questo cursore, ottenendo così risultati aggiornati
  - Dopo la requery(), il cursore è nuovamente attivo

Deprecato
(da api level 11)

- **close()** chiude il cursore, e rilascia i risultati
  - Nonché ogni altra risorsa associata

Aggiornare una query

- Può accadere che, dopo aver fatto una query, si voglia (o si debba
- **deactiva**
  - Ogni te errore
- **requery** cursore, ottenend
  - Dopo l
- **close()**
  - Nonch

L'idea era che i Cursor avessero un ciclo di vita complesso, con la possibilità di sospendere lo scorrimento di un result set, ripetere la query per avere dati aggiornati, e riprendere lo scorrimento.

In realtà, la cosa non ha mai funzionato bene (specialmente in caso di accessi asincroni), e impediva di fare alcune ottimizzazioni sul DBMS, quindi l'intera idea è stata abbandonata.

Oggi la pratica raccomandata è di creare semplicemente un nuovo Cursor quando serve!

Deprecato
(da api level 16)

Deprecato
(da api level 11)

Altre caratteristiche

- SQLite offre alcune altre caratteristiche (che non approfondiamo)
  - Gestione delle transazioni
  - Notifiche associate ai cursori (e relativi listener)
  - Esecuzione asincrona
  - Condivisione di cursori fra processi distinti
    - Anche inviando una *finestra* sui risultati via Parcelable
  - Gestione raffinata degli errori a run-time

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

SQLiteOpenHelper costruttore

- **SQLiteOpenHelper(**
  Context *context*,
  String *name*,
  SQLiteDatabase.CursorFactory *factory*,
  int *version*)
- Come al solito, *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

SQLiteOpenHelper accesso al DB

- Il costruttore di default **non** apre il DB!
  - Si tratta di un tipico pattern *lazy*
    - Non fare fatica finché non è assolutamente indispensabile
- L'Helper offre due metodi di utilità per aprire il DB
  - **getReadableDatabase()** – apre in sola lettura
  - **getWriteableDatabase()** – apre in lettura/scrittura
  - Entrambi restituiscono un SQLiteDatabase
- Solo quando viene chiamato uno dei due metodi di apertura, si usano i parametri del costruttore

SQLiteOpenHelper accesso al DB

- In particolare:
  - Se il DB non esiste, viene invocato l'handler **onCreate()** dell'Helper
  - Se il DB esiste, si legge il suo numero di versione
    - 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

SQLiteOpenHelper accesso al DB

- La sottoclasse **deve** implementare
  - **onCreate()** – qualcuno deve pur decidere lo schema
  - **onUpgrade()** – chissà come si fa l'upgrade
- Gli altri metodi hanno una implementazione di default
  - **onConfigure()** – non fa niente
  - **onOpen()** – non fa niente
  - **onDowngrade()** – lancia un'eccezione
    - Nota: onDowngrade() esiste solo da API Level 11 (Honeycomb, Android 3.0)

Riassunto

- Si crea una sottoclasse di SQLiteOpenHelper per ogni database che usiamo
  - Di solito, solo uno per una App, al limite con tante tabelle dentro
  - La sottoclasse incapsula la logica di creazione e update del DB
- Nella onCreate() dell'Activity, si costruisce un'istanza della sottoclasse con opportuni parametri
  - Questa operazione non è costosa!

Riassunto

- Quando è davvero necessario accedere al DB, si invoca getReadableDatabase() o getWriteableDatabase() sulla sottoclasse
  - La creazione o upgrade può avvenire ora
- I metodi restituiscono un SQLiteDatabase db
- In scrittura, si invocano su db i metodi insert(), update(), ecc.
- In lettura, si invocano su db rawQuery() o query()
  - Queste ultime restituiscono un Cursor cur
  - Su cur si invocano get*Tipo*(*i*) e moveNext()
    - Solitamente in un while (!cur.isAfterLast())

Una vecchia conoscenza

- Ricordate? Fra i tipi di Data Adapter avevamo menzionato il Cursor Adapter
- Un Adapter che prende i dati (da inserire in una View) da un Cursor (ottenuto da un DB)
- Adapter a = new **SimpleCursorAdapter**(

```
    context,
    layout,       // solitamente, R.layout. …
    cur,          // ottenuto da rawQuery() o query()
    from,         // array di nomi di colonne
    to,           // array di ID di TextView nel layout
    flags     );
```

Esercizio

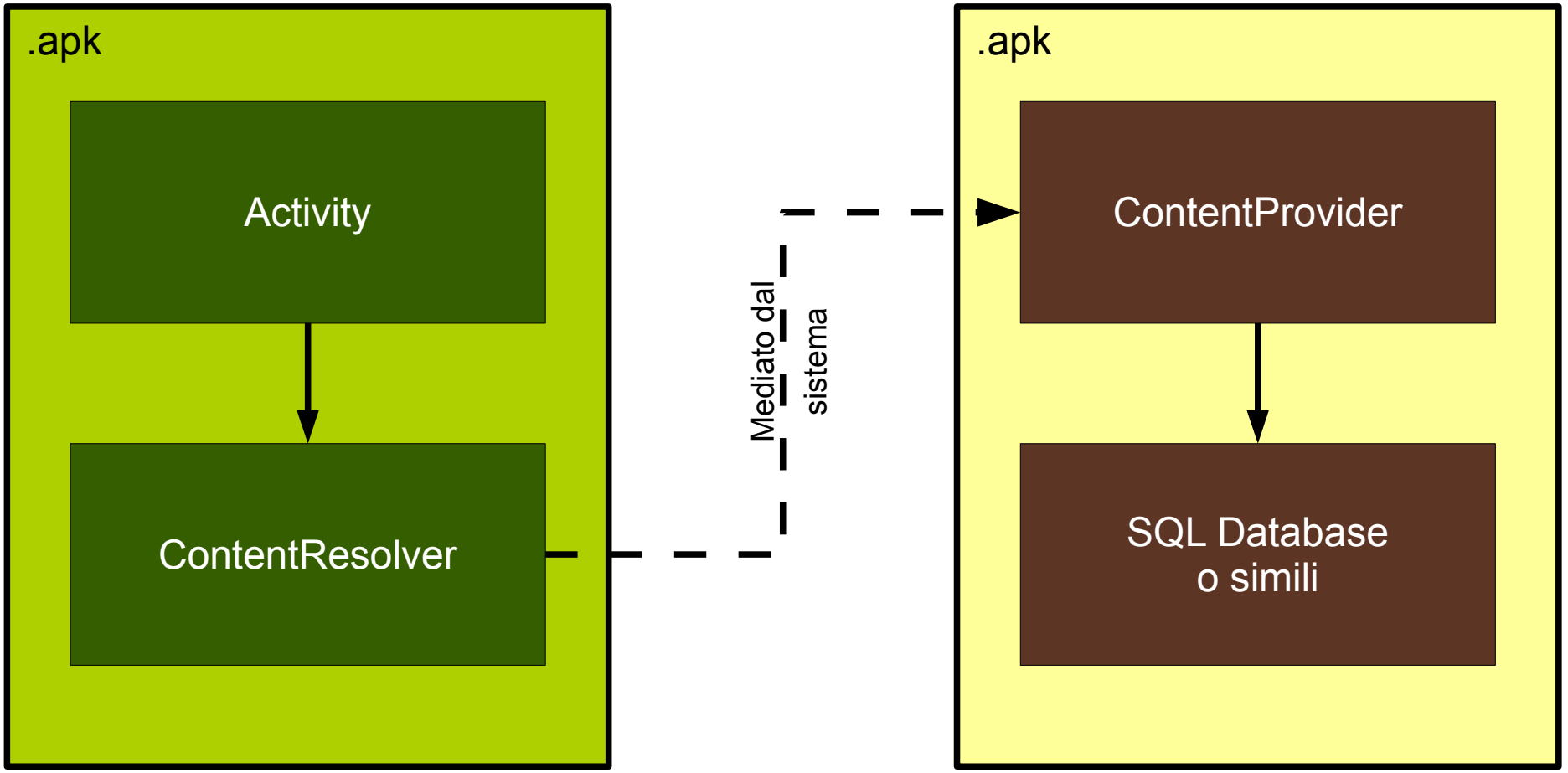
- Scrivere un'app che crei un DB a piacere
  - Meglio se con più tabelle e più campi per tabella
  - Che si presti a una JOIN fra tabelle
- Riempite il DB con valori a caso
- Aggiungete una ListActivity che mostri il risultato di una query al DB
- Quando l'utente fa un long-press su una entry, l'applicazione deve cancellare la riga corrispondente dal DB
  - Suggerimento: servirà un campo chiave. Chiamatelo **_ID**

Content Provider

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
  - Rendere i propri dati accessibili agli altri

Accesso a Content Provider



Leggere dati da un Content Provider

```
ContentResolver cr=getContentResolver();

Uri uri =MediaStore.Images.Media.
          EXTERNAL_CONTENT_URI;
          /* content://media/external/images */

Cursor c=cr.query(uri,
      null,    /* colonne */
      null,    /* selezione */
      null,    /* args */
      null           /* sort */
);

String [] cols = c.getColumnNames();
for (String col: cols) {
    Log.d("TCA",col);
}

Log.d("TCA","# rows = "+c.getCount());
```

```
02-16 18:47:13.327: D/TCA(21998): _id
02-16 18:47:13.327: D/TCA(21998): _data
02-16 18:47:13.327: D/TCA(21998): _size
02-16 18:47:13.327: D/TCA(21998): _display_name
02-16 18:47:13.327: D/TCA(21998): mime_type
02-16 18:47:13.327: D/TCA(21998): title
02-16 18:47:13.327: D/TCA(21998): date_added
02-16 18:47:13.327: D/TCA(21998): date_modified
02-16 18:47:13.327: D/TCA(21998): description
02-16 18:47:13.327: D/TCA(21998): picasa_id
02-16 18:47:13.327: D/TCA(21998): isprivate
02-16 18:47:13.327: D/TCA(21998): latitude
02-16 18:47:13.327: D/TCA(21998): longitude
02-16 18:47:13.327: D/TCA(21998): datetaken
02-16 18:47:13.327: D/TCA(21998): orientation
02-16 18:47:13.327: D/TCA(21998): mini_thumb_magic
02-16 18:47:13.327: D/TCA(21998): bucket_id
02-16 18:47:13.327: D/TCA(21998): bucket_display_name
02-16 18:47:13.327: D/TCA(21998): puid
02-16 18:47:13.327: D/TCA(21998): protect_status
02-16 18:47:13.327: D/TCA(21998): use_count
02-16 18:47:13.327: D/TCA(21998): date_used
02-16 18:47:13.327: D/TCA(21998): rating
02-16 18:47:13.327: D/TCA(21998): width
02-16 18:47:13.327: D/TCA(21998): height
02-16 18:47:13.327: D/TCA(21998): dlna_profile
02-16 18:47:13.327: D/TCA(21998): dlna_share
02-16 18:47:13.327: D/TCA(21998): maker
02-16 18:47:13.327: D/TCA(21998): # rows = 762
```

Leggere dati da un Content Provider

- Una volta ottenuto un Cursor, si procede come di consueto
  - while (!cur.isAfterLast()) { … = cur.getString(3); … }
- **Nota**:
  - Il metodo query() del ContentResolver può richiedere un tempo significativo
  - **MAI** eseguire operazioni lunghe nel thread della UI!
  - La classe di utilità CursorLoader viene in aiuto
    - Caricamento asincrono dei risultati di una query
    - La vedremo successivamente

Leggere dati da un Content Provider

- Per procurarsi l'URI, ci sono vari metodi
  - I Content Provider di sistema offrono spesso costanti predefinite
    - MediaStore.Images.Media.INTERNAL_CONTENT_URI
    - UserDictionary.Words.CONTENT_URI
    - Contacts.People.CONTENT_URI
    - MediaStore.Audio.Albums.EXTERNAL_CONTENT_URI
    - MediaStore.Video.Thumbnails.EXTERNAL_CONTENT_URI
    - VoicemailContract.Vocemails.CONTENT_URI
    - …
  - Per gli altri, ci si affida alla documentazione!

Il formato delle URI

- Le URI usate nei content provider hanno un formato noto
  - Un'intera tabella:
    - **content:**//media/internal/images/

path: identifica la tabella

schema

authority: identifica il provider

  - Uno specifico record:
    - **content:**//media/internal/images/45

id: identifica la riga

Le tabelle esposte da un ContentProvider hanno in genere un campo numerico **_ID** che è la chiave primaria.
La presenza di **_ID** è obbligatoria se si vuole usare il cursor associato a una ListView!

Il formato delle URI

- La classe **android.net.Uri** fornisce metodi di utilità per costruire e convertire URI
- In particolare,
  - Uri Uri.**parse**(String s)
    - data una stringa, costruisce l'Uri relativa
  - Uri Uri.**withAppendedPath**(Uri base, String path)
    - Data una Uri, vi aggiunge in fondo un componente (tipicamente, è il numero di riga)
- Altri metodi sono utili per confrontare o spezzare Uri nelle diverse componenti

Uri, Uri.Builder, ContentUris

- La classe **Uri** rappresenta una URI *immutabile*
  - È efficiente, ma poco flessibile e fa poca validazione
- La classe **Uri.Builder** è un costruttore di URI
  - Progettato per manipolare URI *mutabili*
  - Più metodi per alterare le componenti
  - 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://
  - In particolare, per lavorare con gli _ID

Accesso in scrittura ai dati di un Content Provider

- 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 decidere se e come implementarle
  - In nessun caso si può accedere ai dati sottostanti
- Tipicamente (ma non sempre), le operazioni vengono riflesse su una tabella SQL sottostante

Accesso in scrittura ai dati di un Content Provider

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

```
ContentValues cv = new ContentValues();
cv.put("Name","Vincenzo");
cv.put("Surname","Gervasi");
cv.put("Age", 48);

Uri newrow = cr.insert(uri,cv);
            // il campo _ID è aggiunto automaticamente
```

Uri della tabella

Uri del nuovo record

Accesso in scrittura ai dati di un Content Provider

- Per modificare uno o più record esistenti, si usa il metodo **update()** del ContentResolver
  - Simile a UPDATE … WHERE … in SQL

```
ContentValues cv = new ContentValues();
cv.put("Age", 48);

int n = cr.update(uri, cv, "CF=?", args);
```

- Analogamente per cancellare uno o più record

```
int n = cr.delete(uri, "CF=?", args);
```

Il ruolo dei permessi

- Chi offre un Content Provider **può** richiedere dei permessi nel suo manifesto
  - Con <requires-permission>
- Chi accede a un ContentProvider **deve** usare i permessi nel suo manifesto (se richiesti)
  - Con <uses-permission>
- Lo scopo è duplice:
  - Garantire che chi usa un ContentProvider “lo conosca” a fondo
    - Per esempio, deve sapere lo schema delle tabelle
    - Il nome del permesso fa un po' da “password”
  - Garantire che l'utente approvi i permessi in fase di installazione

Definire un ContentProvider

- Qualunque applicazione può definire un ContentProvider 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

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

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

Definire un ContentProvider

- Si crea una sottoclasse di **ContentProvider**

```
public class ArcobalenoProvider extends ContentProvider {

  @Override
  public int delete(Uri uri, String selection, String[] selectionArgs) { /* ... */ }

  @Override
  public String getType(Uri uri) { /* ... */ }

  @Override
  public Uri insert(Uri uri, ContentValues values) { /* ... */ }

  @Override
  public boolean onCreate() { /* ... */ }

  @Override
  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
  { /* ... */ }

  @Override
  public int update(Uri uri, ContentValues values, String selection,String[] selectionArgs) { /* ... */ }

}
```

Definire un ContentProvider

- La parte della URI comprendente schema e authority è gestita dal sistema
- La parte comprendente il path è a discrezione del provider
  - Ma meglio adeguarsi agli usi standard!
- Data una URI “nostra”, getType() deve restituire il suo tipo MIME
  - vnd.android.cursor.**dir**/rainbow
  - vnd.android.cursor.**item**/rainbow-color

Definire un ContentProvider

- Il nostro ContentProvider avrà il suo ciclo di vita
  - Distinto da quello dell'Activity!
  - Gestito automaticamente dal sistema
    - Viene istanziato quando un ContentResolver deve gestire un'URI la cui authority corrisponde alla nostra
    - Viene chiuso quando non è più necessario
      - Al limite, verrà re-istanziato più avanti
- Il metodo onCreate() deve creare (se non esiste) o aprire (se esiste) il DB corrispondente
  - Solo nel caso (comune) in cui abbiamo un DB!
  - Ricordate **SQLiteOpenHelper**...

Content Provider Riassunto

- Un'applicazione può **definire** un ContentProvider
  - Identificato da un'*authority* (parte di URI)
  - Dichiarato in AndroidManifest.xml (con eventuali permessi)
- Un'applicazione può **usare** un ContentProvider
  - Passa una URI al ContentResolver
  - Quest'ultimo controllo se, nel sistema, è installato un provider in grado di gestire quell'URI
  - Se si, lo istanzia (nel processo del ricevente!) e instrada le richieste query/insert/update/delete

Esempio di uso di Content Provider:

BasicContactables

Esercizio

- Estendere l'esempio **AndroidGallery** illustrato nella lezione sugli Adapter
  - In quel caso, avevamo utilizzato un Adapter grezzo che recuperava per suo conto le immagini da rete
  - Modificarlo in modo che le immagini mostrate dalla Gallery provengano dalle foto dell'utente
    - Ottenute dal ContentProvider relativo!
  - Suggerimento: si ricordi che esistono degli adapter di sistema già pronti per lavorare con i Cursor...

Migliorare l'efficienza

- Compiere molte operazioni con un Content Provider out-of-process può essere costoso
  - Molte operazioni di serializzazione dei dati e IPC
- È possibile anche qui fare *coalescing*
  - Si descrivono le operazioni da fare “in blocco”
  - Si spedisce l'intero pacchetto di operazioni con una sola comunicazione IPC
  - Bonus: l'operazione è *atomica* (= transazioni gratis)
- Classi **ContentProviderOperation** e **ContentProviderOperation.Builder**

Esempio

```
ArrayList<ContentProviderOperation> ops =
          new ArrayList<ContentProviderOperation>();

ops.add(ContentProviderOperation.newInsert(ARCOBALENO_URI)
          .withValue(“Nome”, “Rosso”)
          .withValue(“RGB”,”#FF0000”)
          .build() );

...

ops.add(ContentProviderOperation.newInsert(ARCOB
          .withValue(“Nome”, “Violetto”)
          .withValue(“RGB”,”#9400D3”)
          .build() );

getContentResolver().applyBatch(ARCOBALENO_AUTHORITY, ops);
```

Method chaining

ContentProviderOperation

- **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.

Prende come argomenti una chiave e un valore

Prende come argomento un ContentValues

I Contratti

- 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
  - I provider di sistema definisco spesso contratti complicati, con molte sottoclassi per usi specifici
  - Definiti nel package **android.provider.***

I Contratti - Esempi

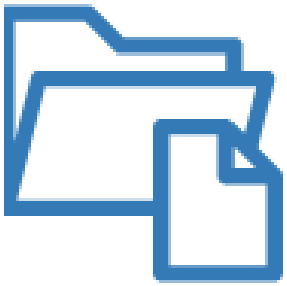
- **CalendarContract**
  - AUTHORITY, CONTENT_URI, ecc.
  - E numerose inner classes, una per ogni tabella
    - Events
      - ORGANIZER, TITLE, EVENT_LOCATION, DTSTART, DTEND, DURATION, ALL_DAY, ...
    - Attendees
      - EVENT_ID, ATTENDEE_NAME, ATTENDEE_EMAIL, ATTENDEE_STATUS, ...
    - Calendars
      - ACCOUNT_NAME, ACCOUNT_TYPE, NAME, CALENDAR_COLOR, ...
    - ecc.

I Contratti - Esempi

- **ContactsContract**
  - AUTHORITY, CONTENT_URI, ecc.
  - E ben *43* inner classes, di tutto di più – alcune hanno a loro volta diverse inner-inner classes
    - Oltre ai dati di base di un contatto, che magari è l'aggregato di più identità, ci sono per esempio informazioni sullo stream di foto provenientei dai suoi social media...
- **CallLog**
  - AUTHORITY, CONTENT_URI
  - Inner classes:
    - Calls – informazioni sulle ultime chiamate ricevute
- **DocumentsContract**, **MediaStore**, **Settings**, **Telephony**, **UserDictionary**, **SyncState**, **SearchRecentSuggestions**, **…**

Il FileProvider

- 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)
- Sarebbe possibile scrivere una sottoclasse di ContentProvider fatta ad-hoc per le nostre esigenze…
- … ma come accade spesso, se l'esigenza è comune anche la soluzione è comune



Il FileProvider

- FileProvider è un Content Provider implementato nella libreria di supporto (da v4 in poi)
- Come tutti i provider, va dichiarato nel Manifest:

```
<manifest     xmlns:android="http://schemas.android.com/apk/res/android"
              package="it.unipi.di.sam.myapp">
   <application ...>
      <provider     android:name="android.support.v4.content.FileProvider"
                    android:authorities="it.unipi.di.sam.myapp.fileprovider"
                    android:grantUriPermissions="true"
                    android:exported="false">
         <meta-data     android:name="android.support.FILE_PROVIDER_PATHS"
                        android:resource="@xml/filepaths" />
      </provider>
      ...
   </application>
</manifest>
```

Il FileProvider

- La struttura delle directory (su fs) associate al FileProvider è definita da un file XML fra le risorse

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

- La “tabella” myimages sarà mappata su «basedir»/it.unipi.di.sam.myapp/files/images/
- Esempio:

content://it.unipi.di.sam.myapp.fileprovider/myimages/default_image.jpg

Altri tipi di path

- Il file XML di specifica dei path può utilizzare vari tipi di *risoluzioni* per un nome di tabella
  - **<files-path … >** - relativo alla sottodirectory files della home dell'app su memoria interna
  - **<cache-path … >** - relativo alla cache dell'app su memoria interna
  - **<external-path … >** - relativo alla root (condivisa) su memoria esterna
  - **<external-files-path … >** - relativo alla root dell'app su memoria esterna
  - **<external-cache-path … >** - relativo alla cache dell'app su memoria esterna
  - **<external-media-path … >** - relativo alla root dell'app in memoria esterna per il tipo di media indicato

FileProvider e permessi

- **file:// … /file**
  - Si applicano i permessi del file system di Linux
  - Difficile controllare quali altre applicazioni abbiano accesso
- **contents://file_provider/ … /file**
  - Si applica il meccanismo dei permessi di Android
  - È possibile creare un Intent che fa riferimento a un URI sul FileProvider, e impostare sull'Intent i flag
    - FLAG_GRANT_URI_READ_PERMISSION o
    - FLAG_GRANT_URI_WRITE_PERMISSION
  - Quando si passa l'Intent a un'altra Activity, quest'ultima avrà il permesso di accedere (in lettura e/o scrittura) al file identificato dall'URI
    - Solo a quello, solo lei, e solo finché l'Activity sta sullo stack

FileProvider e permessi

- 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*) – per uno specifico package
  - revokeUriPermission(*URI*, *flags*) – toglie i diritti a chiunque li abbia avuti, in qualunque modo (anche via Intent): sconsigliato