Lezioni 10-11

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**

Salvataggio temporaneo dello stato

Stato temporaneo

- Il ciclo di vita di un'Activity prevede casi in cui l'istanza della vostra classe può essere eliminata dalla memoria
  - Anche se *logicamente* l'Activity è ancora “viva”
    - Es.: presente nello stack di un task “vivo”, ma non visibile
- In questi casi, è necessario salvare lo **stato transiente** di un'Activity
  - In modo da ripristinarlo più tardi, quando il sistema istanzierà una nuova copia dell'Activity
    - Es.: un'Activity sopra la vostra viene rimossa con Back

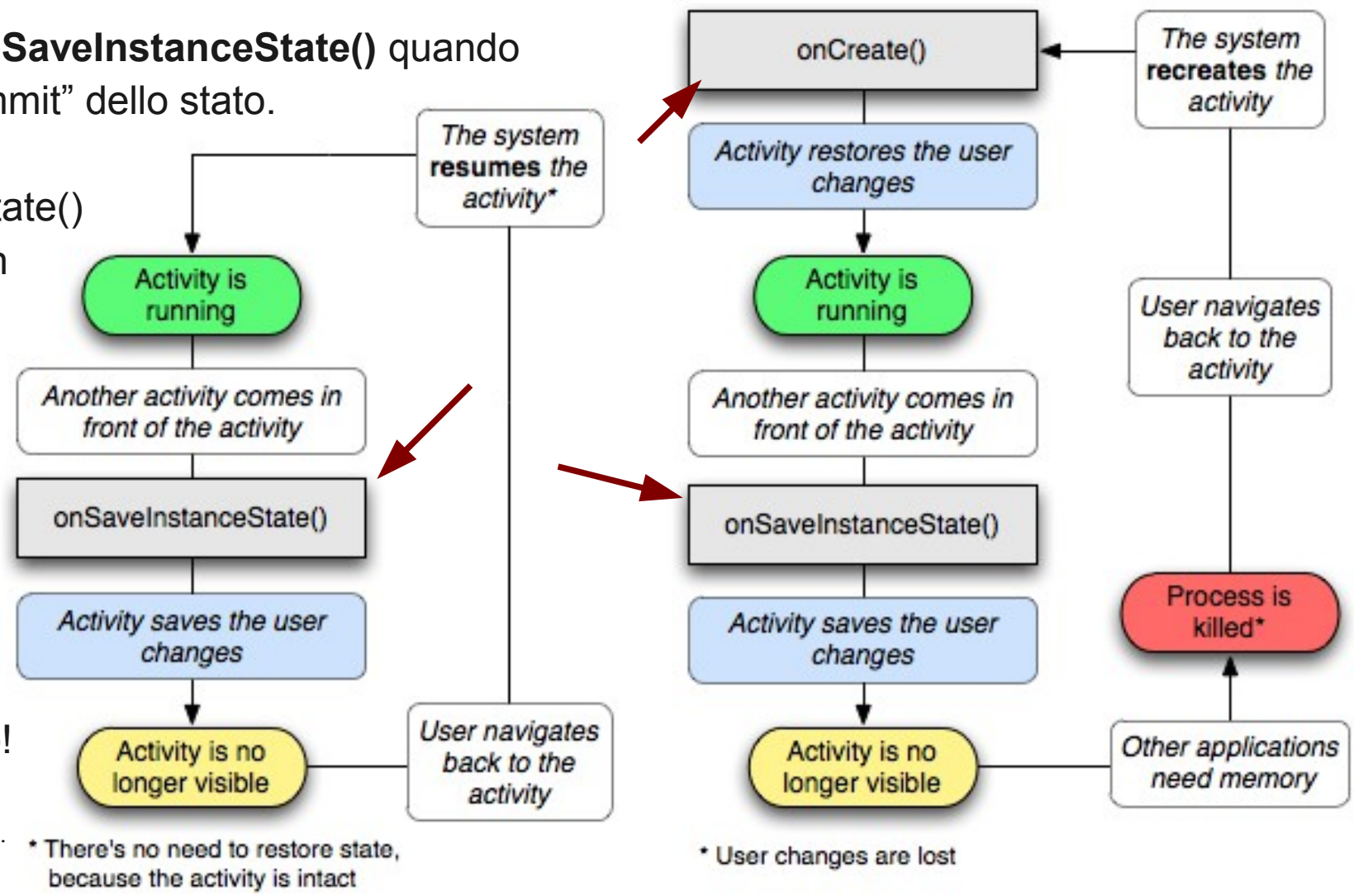
Stato temporaneo

Android invoca **onSaveInstanceState()** quando vuole fare un "commit" dello stato.

onSaveInstanceState() salva lo stato in un **Bundle**.

Se necessario, **onCreate()*** ripristina lo stato dal Bundle.

Può anche non essere necessario!



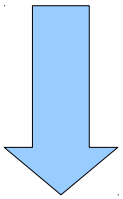
* O onRestoreInstanceState().

Il Bundle

- La classe **Bundle** è una mappa chiave-valore
  - Chiave è una stringa
  - Valore è un **Parcelable**
    - La classe **Parcel** e l'interfaccia **Parcelable** sono usate come meccanismo di IPC in Android
    - Non è serializzazione piena, ma è molto più efficiente
    - In pratica:
      - Tutti i tipi base, array degli stessi, ArrayList, String, altri Bundle, altri oggetti che implementano Parcelable, e oggetti che implementano Serializable
  - Abbondanza di metodi get*Tipo*(key) e put*Tipo*(key,value)

Il Bundle

```
String s = "pippo";
Bundle b = new Bundle();
b.putString("Account",s);
Intent i = new Intent(this, FetchAct.class);
i.putExtras(b);
startActivity(i);
```



```
Intent j = getIntent();
String acc = j.getStringExtra("Account");
```

- Nota:
  - Il Bundle è un meccanismo **generico** per passare valori
  - È possibile istanziare un proprio Bundle, inserire dei valori, e poi metterlo come Extra in un Intent che viene spedito ad altri

Stato temporaneo

- Salvataggio

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

- Ripristino

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

Stato temporaneo

- Il Bundle preparato da onSaveInstanceState() è mantenuto dal sistema...
  - … solo se si prevede di far ripartire **la particolare istanza** dell'Activity
    - Ovvero, se essa viene scaricata per mancanza di memoria
    - **NON** viene chiamato onSaveInstanceState() se l'Activity è terminata, con un Back o con un finish()
  - … in maniera **non permanente**
    - Il Bundle viene tenuto in RAM, in caso di riavvio va tutto perso

Le preferenze

Preferenze & Impostazioni

- Per memorizzare dati in maniera **permanente** si possono usare varie altre tecniche
- Un caso frequente è quello in cui si vogliano memorizzare le **Preferenze** dell'utente
  - Settings, Options, configurazioni, ecc.
  - Si possono usare gli stessi meccanismi anche per memorizzare dati “propri” dell'app, non visibili all'utente
    - es.: data e ora dell'ultimo update, statistiche di utilizzo dell'applicazione
- Android offre un supporto specializzato

SharedPreferences

- Android offre un framework per la gestione generalizzata di preferenze
- La classe **SharedPreference** rappresenta una mappa chiavi-valori
  - Simile al Bundle, ma con alcune importanti differenze:
    - Le preferenze sono memorizzate in maniera **permanente**
    - I valori possono essere solo **tipi base**, **String** o **Set<String>**
    - Il sistema gestisce l'**aggiornamento atomico** (commit)
    - È disponibile un sistema di **notifiche** per essere avvisati quando le preferenze cambiano
    - Le preferenze possono essere **per-Activity** o **globali**

SharedPreferences

- Le preferenze vengono salvate su un file
- Se ne possono avere molte!
  - Il programmatore può dare un *nome* a un insieme di preferenze per l'App
  - Oppure, ogni Activity può usare le proprie
    - In pratica: il nome della classe diventa il nome delle preferenze
- Path sul file system:
  - /data/data/*package*/shared_prefs/*nome*.xml
  - /data/data/*package*/shared_prefs/*package*_preferences.xml

Area dati dell'app identificata dal nome del *package*

SharedPrefs di default

SharedPreferences

- A differenza che nei Bundle, nel caso delle preferenze le scritture sono **transazionali**
  - Prima viene fatto un insieme di aggiornamenti
  - Poi si effettua il **commit** che le scrive tutte insieme
- Ciò garantisce:
  - **Consistenza**: non può capitare che alcuni campi vengano aggiornati e altri no; il commit è atomico
  - **Coalescing** delle notifiche: chi è in attesa viene notificato una volta sola per l'intero insieme di modifiche, non ad ogni campo cambiato

Preferences Editor

- Le modifiche alle preferenze vengono fatte tramite un **editor** (che viene ottenuto dalle preferenze stesse)
- L'**editor** offre i metodi **put*Tipo*(chiave, valore)** per inserire coppie chiave-valore nella sua tabella temporanea
- Per copiare la tabella temporanea sulle preferenze sono disponibili due metodi dell'editor
  - **commit()** – aggiunge la tabella dell'editor a quella delle preferenze, e salva immediatamente su disco
    - Sicura, in caso di errore restituisce un codice d'errore, meno efficiente
  - **apply()** – aggiunge la tabella dell'editor a quella delle preferenze in memoria, e schedula la scrittura asincrona del risultato su disco
    - Meno sicura, non verifica gli errori, più efficiente

Scrittura di preferenze

- Ottenere un oggetto SharedPreferences
  - Dotato di nome: **getSharedPreferences(*nome*,*modo*)**
    - Metodo di Context
    - Il *modo* può essere
      - MODE_PRIVATE
      - MODE_WORLD_READABLE
      - MODE_WORLD_WRITEABLE
      - MODE_MULTI_PROCESS (in OR con i precedenti)
  - Privato: **getPreferences(*modo*)**
    - Metodo di Activity
    - Il *modo* può essere come sopra, tranne MULTI_PROCESS

Scrittura di preferenze

- Ottenere un oggetto SharedPreferences
  - Di default per un dato Context: **PreferenceManager.getDefaultSharedPreferences(*context*)**
    - Il context può essere vario
      - Context.getApplicationContext() → context dell'applicazione intera
      - Activity → context della specifica activity
      - Service, BroadcastReceiver, ContentProvider → idem
      - e *molti* altri!
        - Wallpaper animati, metodo di input (=tastiera virtuale), agente di backup, home screen, spell checker di sistema, ecc.

Scrittura di preferenze

- Scrittura

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

- Lettura

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

Default da usare se la chiave non esiste

Notifiche sulle preferenze

- Le SharedPreferences possono essere (ovviamente) condivise fra più activity
- Ciascuna di questa può scriverci dentro
  - Ricordate: il commit è atomico
- Le altre activity possono voler essere informate dei cambiamenti apportati alle preferenze
  - Ricordate: si chiamano preferenze, ma possono essere dati qualunque, anche interni!
    - Purché esprimibili con tipi base, String, Set<String>

Notifiche sulle preferenze

- Si usa il solito meccanismo dei **Listener**
  - prefs.**registerOnSharedPreferenceChangeListener(*listener*)**
- Il listener deve implementare l'interfaccia **OnSharedPreferencesChangeListener**
  - Ha un unico metodo: **onSharedPreferenceChanged(*prefs*, *chiave*)**
    - No comment sui nomi...
    - La *chiave* può essere stata aggiunta, modificata o rimossa (e quindi non esistere più in *prefs*!)
- Per rimuovere un listener, ***ovviamente*** si chiama
  - prefs.**unregisterOnSharedPreferenceChangeListener(*listener*)**

Gestione strutturata delle preferenze

- I metodi che abbiamo visto sono relativamente semplici e piuttosto comodi per salvare un po' di dati generici
- Tuttavia, quando le preferenze rappresentano davvero preferenze dell'utente, e devono essere editabili, le cose si complicano
  - Serve un'Activity per fornire la GUI
  - Vanno gestiti tutti gli eventi relativi
- Android fornisce però un framework ad-hoc!

PreferenceScreen

- Note quali preferenze servono, si può costruire la GUI corrispondente in maniera sistematica
- Conviene allora definire le preferenze in maniera **dichiarativa** e lasciar fare al sistema
  - In pratica... tramite un file XML
- L'editing, l'aggiornamento, le notifiche, l'undo (con Back) ecc. sono gestiti autonomamente
- All'applicazione non resta da fare altro che registrare un listener (e reagire)!

PreferenceScreen

- Il file che descrive le preferenze va in res/xml
  - **Non** è un layout, anche se il sistema deriva un layout
- L'elemento radice è **<PreferenceScreen>**
- All'interno, può contenere
  - **<PreferenceScreen>** – struttura gerarchica
  - **<PreferenceCategory>** – raggruppa opzioni correlate
  - **<CheckBoxPreference>**, **<EditTextPreference>**, **<ListPreference>**, **<RingtonePreference>**
  - Altri editor di preferenze custom
    - Sottoclassi di Preference o dei precedenti

PreferenceScreen

- I nodi hanno alcuni attributi standard
  - **android:key** – la chiave di questa entry nelle pref
  - **android:title** – il titolo del'entry
  - **android:summary** – la descrizione dell'entry
  - **android:icon** – un riferimento a risorsa per l'icona
  - **android:defaultValue** – valore di default dell'entry
- Altri sono specifici di certi tipi, o di uso più raro
  - es.: ordinamento, subordinazione all'attivazione di altre entry, layout custom

PreferenceScreen esempio: prefs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >

   <PreferenceCategory android:title="Audio" >
      <CheckBoxPreference
         android:defaultValue="true"
         android:key="audio_in"
         android:summary="Abilita la registrazione dell'audio dal microfono incorporato."
         android:title="Attiva microfono" />
      <CheckBoxPreference
         android:defaultValue="true"
         android:key="audio_out"
         android:summary="Abilita la riproduzione dell'audio dall'altoparlante incorporato."
         android:title="Attiva altoparlante" />

      <ListPreference
         android:dialogTitle="Scegli voce"
         android:entries="@array/voci"
         android:entryValues="@array/valori"
         android:key="Voce"
         android:summary="Puoi scegliere che tipo di voce utilizzare"
         android:title="Tipo voce" />
   </PreferenceCategory>
```

PreferenceScreen esempio: prefs.xml

```
<PreferenceCategory android:title="Video" >
    <CheckBoxPreference
        android:defaultValue="true"
        android:key="video"
        android:summaryOff="Abilita il video (occupa lo schermo)"
        android:summaryOn="Disabilita il video (solo audio)"
        android:title="Abilita il video" />

    <EditTextPreference
        android:defaultValue="Roboto"
        android:key="nome_pg"
        android:summary="Puoi scegliere il nome da dare al tuo personaggio"
        android:title="Nome personaggio" />

    <PreferenceScreen android:title="Avanzate..." >

            ...

    </PreferenceScreen>
</PreferenceCategory>

</PreferenceScreen>
```

La PreferenceActivity

- La classe PreferenceActivity si occupa di
  - leggere il nostro prefs.xml
  - generare al volo un layout adeguato
  - interagire con l'utente
    - Anche in maniera complessa: navigazione in sotto-schermi, dialog, ecc.
  - salvare le impostazioni nelle SharedPrefs alla fine
    - Il che può far partire delle notifiche

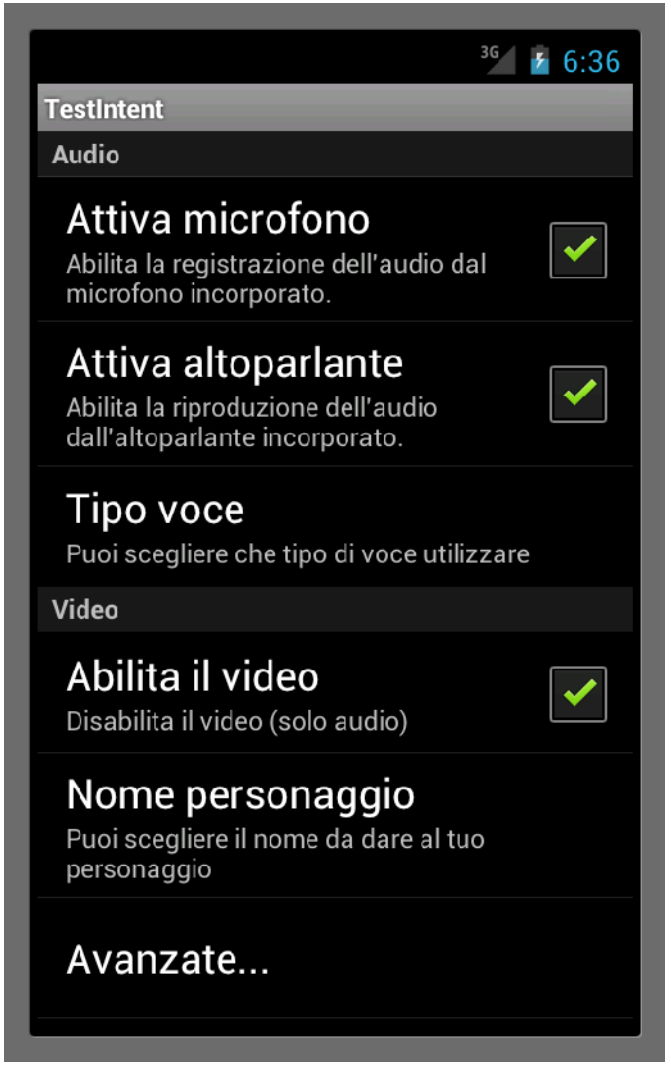
La PreferenceActivity esempio

```
public class TestIntentActivity extends PreferenceActivity {

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.prefs);
   }
}
```

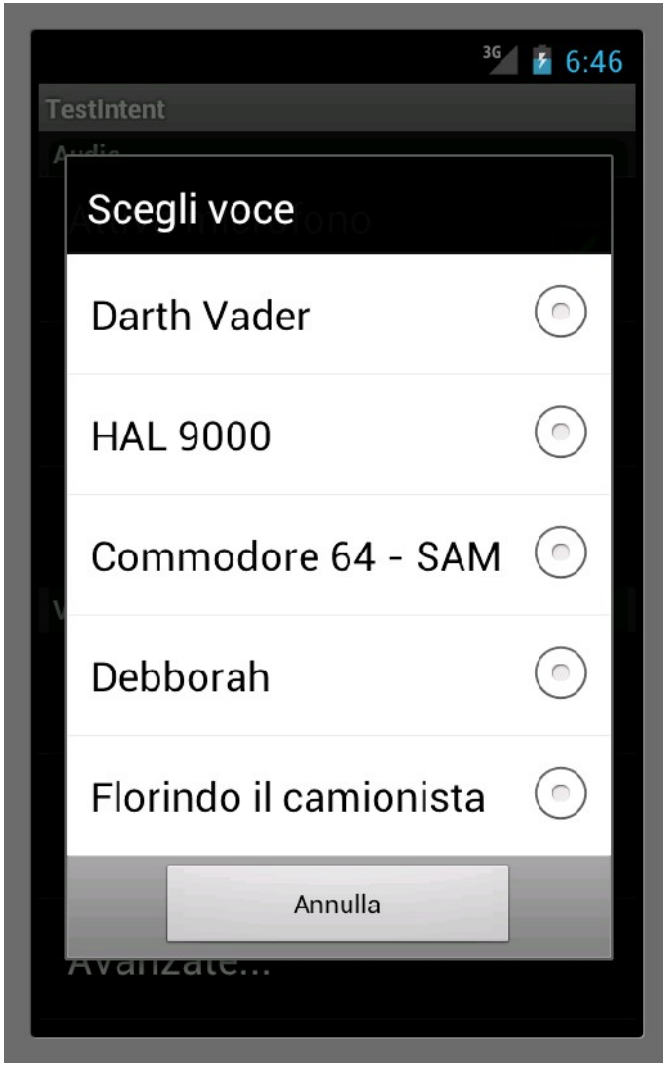
- Non si può dire che ci si ammazzi di lavoro...
- La classe può opzionalmente implementare un change listener, e reagire opportunamente
  - O, meglio, lasciare che si registrino le altre activity

La PreferenceActivity esempio

- L'activity parte, presenta la familiare interfaccia dei settings all'utente
- Come bonus ulteriore, si adatta automaticamente al tema corrente
- L'uso di riferimenti a risorse nel file XML rende anche il tutto facilmente localizzabile
  - Noi non l'abbiamo fatto, ma...



La PreferenceActivity esempio



- Anche la gestione dei dialog è completamente invisibile
- I valori scelti però verranno scritti alla fine nelle preferenze
- Vengono usate sempre le **preferenze globali dell'applicazione**

```
Context c=getApplicationContext();
SharedPreferences prefs=
  PreferenceManager.
  getDefaultSharedPreferences(c);
```

Condividere preferenze

- Si possono invocare dalle proprie preferenze quelle di sistema (o di altre app)

In prefs.xml

```
<PreferenceScreen android:title="Avanzate..." >
   <intent android:action="android.settings.SOUND_SETTINGS"  />
</PreferenceScreen>
```

- Si possono rendere le proprie preferenze invocabili da altre app (già visto!)

In AndroidManifest.xml

```
<activity
   android:name=".TestIntentActivity"
   android:label="@string/app_name" >
   <intent-filter>
      <action android:name="it.unipi.di.sam.PG_PREFS"/>
   </intent-filter>
</activity>
```

Preferenze di sistema

ACTION_ACCESSIBILITY_SETTINGS	Activity Action: Show settings for accessibility modules.
ACTION_ADD_ACCOUNT	Activity Action: Show add account screen for creating a new account.
ACTION_AIRPLANE_MODE_SETTINGS	Activity Action: Show settings to allow entering/exiting airplane mode.
ACTION_APN_SETTINGS	Activity Action: Show settings to allow configuration of APNs.
ACTION_APPLICATION_DETAILS_SETTINGS	Activity Action: Show screen of details about a particular application.
ACTION_APPLICATION_DEVELOPMENT_SETTINGS	Activity Action: Show settings to allow configuration of application development-related settings.
ACTION_APPLICATION_SETTINGS	Activity Action: Show settings to allow configuration of application-related settings.
ACTION_BLUETOOTH_SETTINGS	Activity Action: Show settings to allow configuration of Bluetooth.
ACTION_DATA_ROAMING_SETTINGS	Activity Action: Show settings for selection of 2G/3G.
ACTION_DATE_SETTINGS	Activity Action: Show settings to allow configuration of date and time.
ACTION_DEVICE_INFO_SETTINGS	Activity Action: Show general device information settings (serial number, software version, phone number, etc.).
ACTION_DISPLAY_SETTINGS	Activity Action: Show settings to allow configuration of display.
ACTION_INPUT_METHOD_SETTINGS	Activity Action: Show settings to configure input methods, in particular allowing the user to enable input methods.
ACTION_INPUT_METHOD_SUBTYPE_SETTINGS	Activity Action: Show settings to enable/disable input method subtypes.
ACTION_INTERNAL_STORAGE_SETTINGS	Activity Action: Show settings for internal storage.
ACTION_LOCALE_SETTINGS	Activity Action: Show settings to allow configuration of locale.
ACTION_LOCATION_SOURCE_SETTINGS	Activity Action: Show settings to allow configuration of current location sources.
ACTION_MANAGE_ALL_APPLICATIONS_SETTINGS	Activity Action: Show settings to manage all applications.
ACTION_MANAGE_APPLICATIONS_SETTINGS	Activity Action: Show settings to manage installed applications.
ACTION_MEMORY_CARD_SETTINGS	Activity Action: Show settings for memory card storage.
ACTION_NETWORK_OPERATOR_SETTINGS	Activity Action: Show settings for selecting the network operator.
ACTION_NFCSHARING_SETTINGS	Activity Action: Show NFC sharing settings.
ACTION_PRIVACY_SETTINGS	Activity Action: Show settings to allow configuration of privacy options.
ACTION_QUICK_LAUNCH_SETTINGS	Activity Action: Show settings to allow configuration of quick launch shortcuts.
ACTION_SEARCH_SETTINGS	Activity Action: Show settings for global search.
ACTION_SECURITY_SETTINGS	Activity Action: Show settings to allow configuration of security and location privacy.
ACTION_SETTINGS	Activity Action: Show system settings.
ACTION_SOUND_SETTINGS	Activity Action: Show settings to allow configuration of sound and volume.
ACTION_SYNC_SETTINGS	Activity Action: Show settings to allow configuration of sync settings.
ACTION_USER_DICTIONARY_SETTINGS	Activity Action: Show settings to manage the user input dictionary.
ACTION_WIFI_IP_SETTINGS	Activity Action: Show settings to allow configuration of a static IP address for Wi-Fi.
ACTION_WIFI_SETTINGS	Activity Action: Show settings to allow configuration of Wi-Fi.
ACTION_WIRELESS_SETTINGS	Activity Action: Show settings to allow configuration of wireless controls such as Wi-Fi, Bluetooth and Mobile networks.

Accesso al file system

Il file system di Android

- Android è uno strato “sopra” Linux
- Il file system vero e proprio è quindi gestito interamente da quest'ultimo
  - File, directory, hard/soft-link, diritti, proprietario, ecc.
- La gestione dei file di Android è dunque uno strato (interfaccia Java) “sopra” il file system vero
- Accedere direttamente ai file su Android **è raro**
  - **Si usano più spesso le preferenze o i database**
  - **Oppure, in sola lettura, le risorse (raw) e gli asset**

java.io.* e java.nio.*

- È disponibile la gestione dei file standard di Java tramite le classi dei package
  - java.io – accesso ai file tradizionale
  - java.nio – accesso asincrono ai file
- Sono poi disponibili tutte le consuete facility
  - Wrapper di vario tipo
    - Reader/Writer, Buffered(Input/Output)Stream, Object(Input/Output)Stream e la serializzazione, StringReader, StringTokenizer, ecc.
  - Navigazione con gli oggetti File (= file e directory)

Pecularità di Android

- Tuttavia, Android presenta delle peculiarità:
  - L'utente **non siete voi**, ma la vostra App
    - Ogni app è un utente diverso, i file sono mutuamente segregati in directory distinte
    - Per default, i file sono “privati” all'app (ma possono essere resi pubblici)
  - I dispositivi distinguono **memoria “interna”** (al telefono) e **memoria “esterna”** (scheda SD e simili)
  - L'ambiente definisce **posti standard** in cui memorizzare vari tipi di dato condivisi
    - Musica, video, ebook, suonerie, foto, podcast, ...

Pecularità di Android

- Ogni App ha una **base directory** in **memoria interna** e una in **memoria esterna** (se c'è)
  - Queste directory vengono cancellate se l'app viene disinstallata
  - L'App può crearvi file e sotto-directory a piacere
- Per certi usi, sono definiti **nomi convenzionali**
  - In questo modo, il Media Scanner di sistema può trovare e classificare i dati multimediali
- L'App ha una **cache directory** per i file temporanei
  - Il sistema la cancella se ha bisogno di spazio – ma meglio limitarsi!
- Il sistema fornisce una **directory condivisa** per dati pubblici
  - Nuovamente, all'interno si usano i nomi convenzionali

File in memoria interna

- Il Context offre metodi di utilità per accedere ai file nella base dir interna
  - FileOutputStream **openFileOutput(*nome*, *modo*)**
  - FileInputStream **openFileInput(*nome*)**
  - String [] **fileList()**
  - void **deleteFile(*nome*)**
- E altri metodi per gestire le directory
  - File **getFilesDir()** – restituisce il path alla base dir
  - File **getDir(*nome*, *modo*)** – apre o crea una sottodir
  - File **getCacheDir()** – restituisce il path alla cache dir

File in memoria esterna

- La scheda SD potrebbe mancare, o essere stata estratta, oppure montata via USB su un PC
- Prima di accedere alla memoria esterna, è bene controllare che sia presente
  - String **getExternalStorageState()**
  - Valori restituiti: costanti stringa definite in Environment
  - I file sono accessibili se viene restituto
    - Environment.MEDIA_MOUNTED – tutto ok!
    - Environment.MEDIA_MOUNTED_READ_ONLY – solo lettura

Metodo (statico) di **Environment**

File in memoria esterna

MEDIA_BAD_REMOVAL	if the media was removed before it was unmounted.
MEDIA_CHECKING	if the media is present and being disk-checked
MEDIA_MOUNTED	if the media is present and mounted at its mount point with read/write access.
MEDIA_MOUNTED_READ_ONLY	if the media is present and mounted at its mount point with read only access.
MEDIA_NOFS	if the media is present but is blank or is using an unsupported filesystem
MEDIA_REMOVED	if the media is not present.
MEDIA_SHARED	if the media is present not mounted, and shared via USB mass storage.
MEDIA_UNMOUNTABLE	if the media is present but cannot be mounted.
MEDIA_UNMOUNTED	if the media is present but not mounted.

File in memoria esterna

- File getExternalFilesDir(*tipo*) – restituisce la directory in cui l'App dovrebbe salvare i file del *tipo* indicato
  - Tipicamente, /Android/data/*package*/files/...
  - Tuttavia, la memoria esterna è condivisa, non ci sono diritti né protezioni: è solo una convenzione
- Se il *tipo* è **null**, viene restituita la base dir della vostra app
- La base dir e tutto il contenuto vengono cancellati se l'app viene disinstallata

File in memoria esterna

DIRECTORY_ALARMS	Standard directory in which to place any audio files that should be in the list of alarms that the user can select (not as regular music).
DIRECTORY_DCIM	The traditional location for pictures and videos when mounting the device as a camera.
DIRECTORY_DOWNLOADS	Standard directory in which to place files that have been downloaded by the user.
DIRECTORY_MOVIES	Standard directory in which to place movies that are available to the user.
DIRECTORY_MUSIC	Standard directory in which to place any audio files that should be in the regular list of music for the user.
DIRECTORY_NOTIFICATIONS	Standard directory in which to place any audio files that should be in the list of notifications that the user can select (not as regular music).
DIRECTORY_PICTURES	Standard directory in which to place pictures that are available to the user.
DIRECTORY_PODCASTS	Standard directory in which to place any audio files that should be in the list of podcasts that the user can select (not as regular music).
DIRECTORY_RINGTONES	Standard directory in which to place any audio files that should be in the list of ringtones that the user can select (not as regular music).

Anche in questo caso, si tratta di costanti definite nella classe **Environment**

File condivisi e cache in memoria esterna

- Un'applicazione che voglia esplicitamente **condividere file** (tipicamente, media), può salvarli nella directory condivisa di sistema
  - File **getExternalStoragePublicDirectory(*tipo*)**
- Questi file **non** vengono cancellati quando l'applicazione viene disinstallata
- Se l'app ha bisogno di una **cache ampia**, può utilizzare la memoria esterna
  - File **getExternalCacheDir()**
- La cache esterna verrà svuotata alla disinstallazione
  - Ma non in caso di SD piena – la gestione è a carico vostro!

Accesso a Database

SQLite

- Android incorpora una versione di SQLite
  - Database di uso generale
  - Particolarmente “piccolo”
  - 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 istruzione 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 … on è (affat…
- SQLite … anziche… o specifi…
  - **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[]

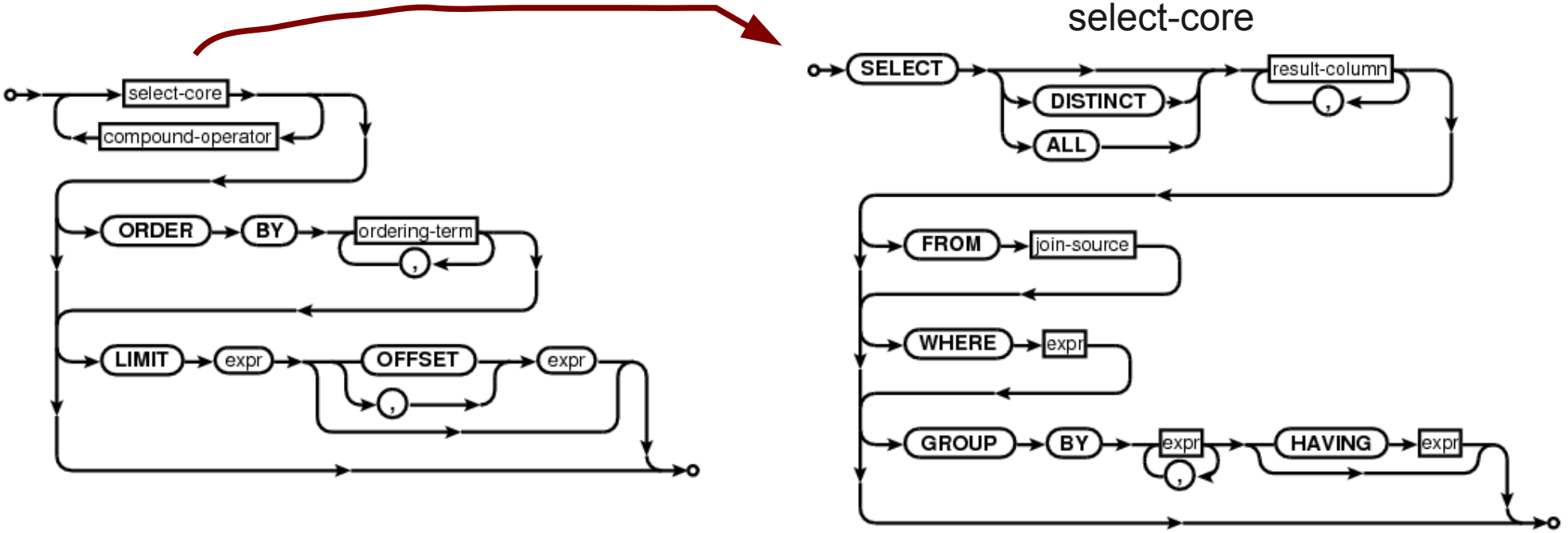
Esempio – UPDATE

```
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

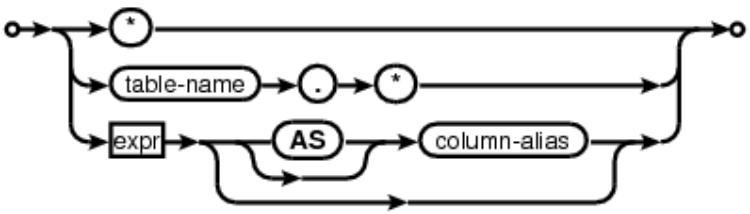
Eseguire una SELECT

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



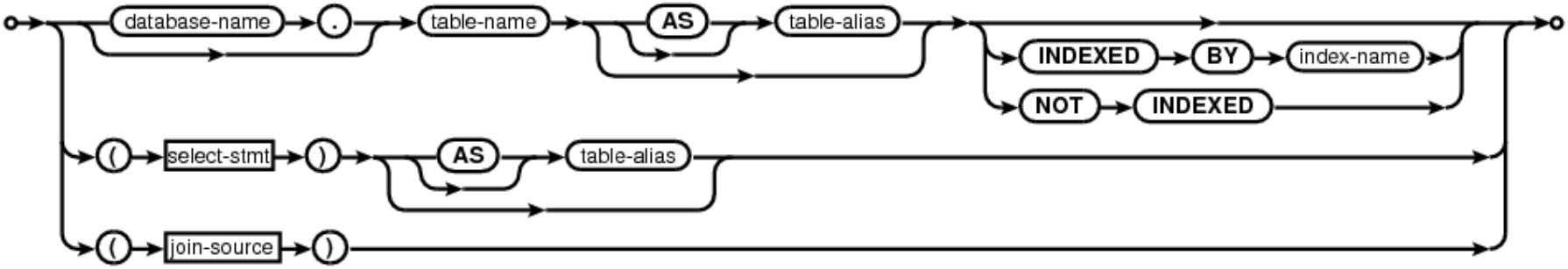
Eseguire una SELECT

result-column

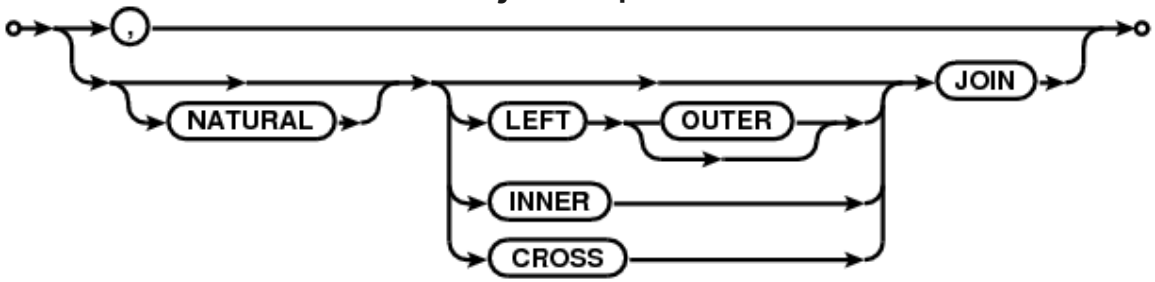


join-source

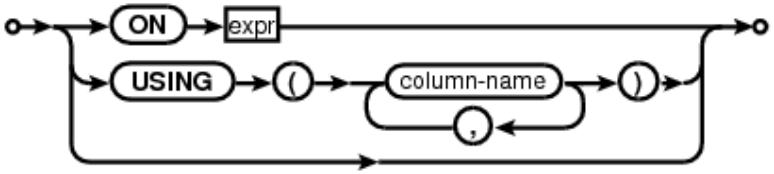
single-source



join-op

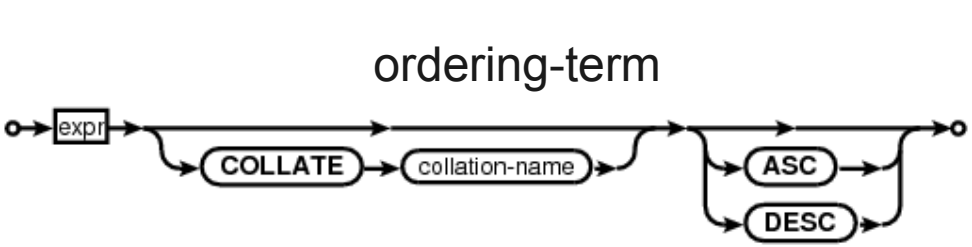


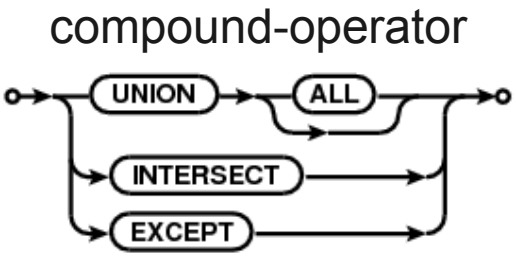
join-constraint



Grafici di sintassi da www.sqlite.org

Eseguire una SELECT





- 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

Deprecato
(da api level 16)

  - Ogni tentativo di usare un cursore disattivato da errore
- **requery()** ripete la query originale di questo cursore, ottenendo così risultati aggiornati

Deprecato
(da api level 11)

  - Dopo la requery(), il cursore è nuovamente attivo
- **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 (…zione
- **deactiv…** → Deprecato (da api level 16)
  - Ogni …to da errore
- **requer…esto** cursore…ati → Deprecato (da api level 11)
  - Dopo… attivo
- **close()** …nsultati
  - Nonché ogni altra risorsa associata

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!

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
  - **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**