Persistence

How to store data…

Preview

This is the contain we'll see :

- Presentation
- Instance State
- Shared Preferences
- SQLite

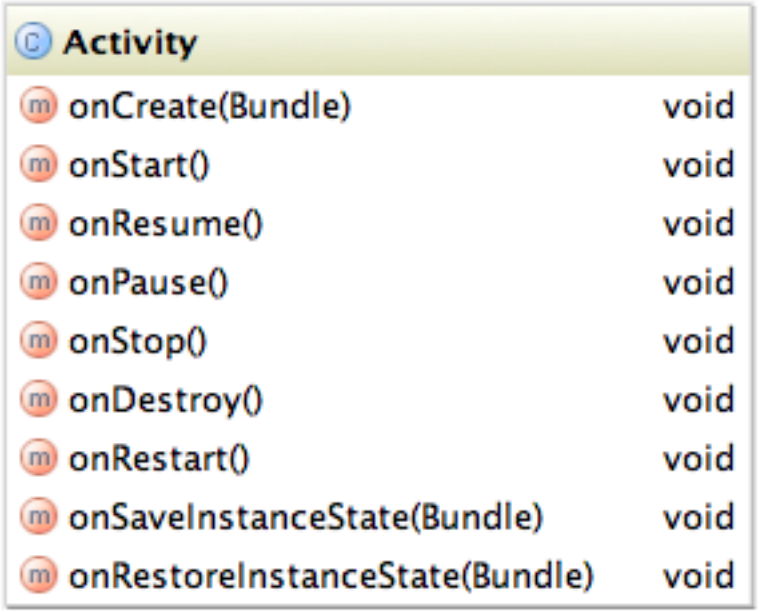
Presentation

- Android provides four ways to store data
  - Instance State
  - Shared Preferences
  - SQLite databases
  - Files
- We’re going to see the first three.

Instance State

- You have seen earlier an activity's lifecycle
- A background activity can be unloaded if another needs memory
- How to save an activity state to allow the user to retrieve the activity's previous state ?
  - Thanks to Instance State !
- We're going to see the two activity methods to manage instance state :

onSaveInstanceState(…)
onRestoreInstanceState(…)



Instance State

- In order for the Android system to restore the state of the views in your activity, **each view must have a unique ID**, supplied by the android:id attribute

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Instance State

- **onSaveInstanceState(Bundle)**
  - Called to retrieve per-instance state from an activity before being killed so that the state can be restored in onCreate(Bundle) or onRestoreInstanceState(Bundle) (the Bundle populated by this method will be passed to both).
- **onRestoreInstanceState(Bundle)**
  - This method is called after onStart() when the activity is being re-initialized from a previously saved state.

Instance State

- By default, Instance State saves the values of all views *with **id** attribute*
- If you want to save more information, just override the two methods we have just seen

```
private String myInformation;
...

protected void onSaveInstanceState(Bundle outState) {
    outState.putString("anotherInformation", myInformation);
    super.onSaveInstanceState(outState);
}

protected void onRestoreInstanceState(Bundle savedInstanceState){
    super.onRestoreInstanceState(savedInstanceState);
    myInformation =

       savedInstanceState.getString("anotherInformation");
}
```

Shared Preferences

- Shared across all components in an application
- Set of key/value pairs
- Can only store **boolean**, **int**, **long**, **float** and **String** values
- Permission can be given :
  - **MODE_PRIVATE**
    - Default value, the created file is only accessible by the application that created it.
  - **MODE_WORD_READABLE**
    - Other applications can read the file but not modify it.
  - **MODE_WORD_WRITABLE**
    - Other applications can modify the file.

Shared Preferences

- Examples :
  - Retrieve shared preferences :
  - getPreferences(..) : default preferences file for an activity – no need for a filename !

```
SharedPreferences prefs =

      getPreferences(Context.MODE_PRIVATE);

// If there is no value for "username", return null
String username = prefs.getString("username", null);

// If there is no value for "isAdmin", return false
boolean admin = prefs.getBoolean("isAdmin", false);

// If there is no value for "id", return zero
long id = prefs.getLong("id", 0L);
```

Shared Preferences

- Examples :
  - Save **shared** preferences :

getSharedPreferences(String filename, int mode)

```
SharedPreferences prefs =
              getSharedPreferences("MYPREFS",
Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();

editor.putString("username", "Droid");
editor.putBoolean("isAdmin", true);
editor.commit();
```

SQLite Databases

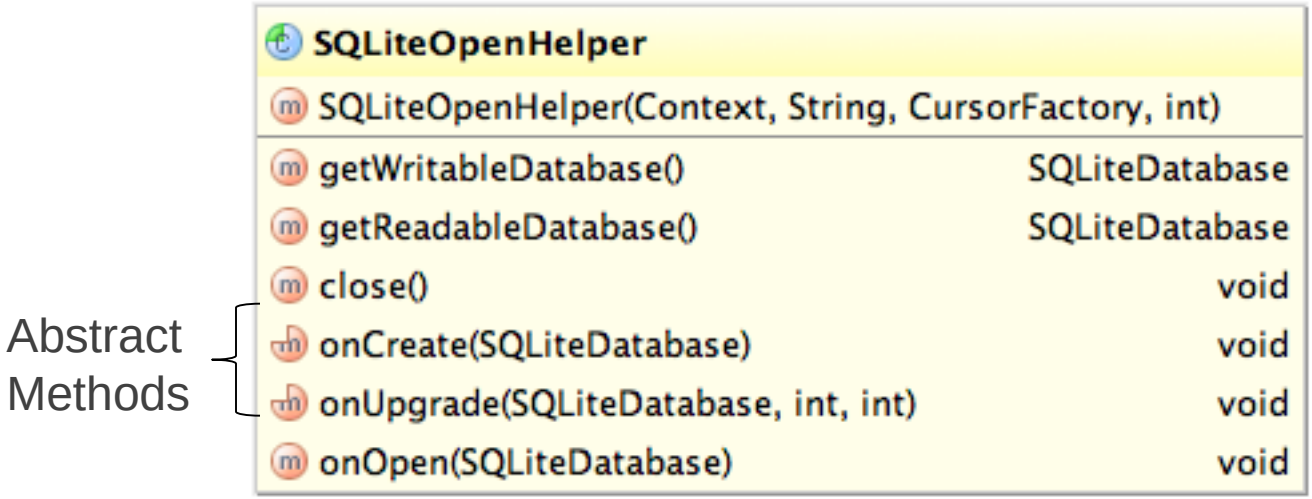
- Relational Database Management System
- Useful to stock complex data
- Each database is dedicated to only one application
- An application can have several databases

SQLite Databases

- Don’t design your SQLite database as a MySQL or PostgreSQL ones
- Mobile devices are not dedicated database servers
  - Little storage space
  - Little memory
- Store only what you need
- Avoid frequent requests
- Design SQLite databases with :
  - A simple structure
  - Data easily identifiable
- Don’t store binary data !

SQLiteOpenHelper

- To simplify your code to create or update a Database schema, the SDK propose you a Helper class named : **SQLiteOpenHelper**.
- To use it, create your proper class and extend it.



SQLiteOpenHelper

- Example :

```
public class MyOpenHelper extends SQLiteOpenHelper {

   private static final String DATABASE_NAME = "my.db";
   private static final int DATABASE_VERSION = 2;
   private static final String TABLE_NAME = "persons";
   private static final String TABLE_CREATE =
       "CREATE TABLE " + TABLE_NAME + " (" +
       "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
       "name TEXT NOT NULL);";

   public MyOpenHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

...
```

SQLiteOpenHelper

- Example :

```
...

   public void onCreate(SQLiteDatabase db) {
       db.execSQL(TABLE_CREATE);
   }

   public void onUpgrade(SQLiteDatabase db,
                                                    int
oldVersion, int newVersion) {

       Log.w("Example", "Upgrading database, this will drop"
                                                      +
"tables and recreate.");

       db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
       onCreate(db);
   }

}
```

SQLiteOpenHelper

- This class provides two other useful methods
  - **SQLiteDatabase getWritableDatabase()**
    - Return a SQLiteDatabase instance to read or write in the Database. Throw an exception if the database cannot be opened for writing (bad permission or full disk).
  - **SQLiteDatabase getReadableDatabase()**
    - Return a SQLiteDatabase instance with read-only access to the database.
  - Both will create the database if it doesn't exist.

SQLiteDatabase

- Exposes methods to manage a SQLite database
- Has methods to create, delete, execute SQL commands, and perform other common database management tasks
- We're going to see some useful methods :
  - **void execSQL(...)**
  - **long insert(…)**
  - **int update(…)**
  - **int delete(…)**
  - **Cursor query(…)**

SQLiteDatabase

- **void execSQL(String sql) :**
  - Execute a single SQL statement that is not a query
    - For example, CREATE TABLE, DELETE, INSERT, etc.
  - Example :

```
SQLiteDatabase db = ...

db.execSQL("DROP TABLE IF EXISTS my_table");
```

SQLiteDatabase

- **long insert (String table, String nullColumnHack, ContentValues values) :**
  - Convenience method for inserting a row into the database
  - Three parameters :
    - **table :** The table to insert the row into
    - **nullColumnHack :**
      - SQL doesn't allow inserting a completely empty row
      - If initial values are empty this column will explicitly be assigned a NULL value
    - **values :**
      - Map containing the column values for the row
        - The keys should be the column names
        - The values the column values

SQLiteDatabase

- **long insert (String table, String nullColumnHack, ContentValues values) :**
  - Return the row ID of the inserted row
  - Example :

```
SQLiteDatabase db = ...

ContentValues values = new ContentValues();
values.put("name", "Cartman");

db.insert("persons", null, values);
```

SQLiteDatabase

- **int update (String table, ContentValues values, String whereClause, String[] whereArgs) :**
  - Convenience method for updating rows in the database
  - Four parameters :
    - **table :** the table to update in
    - **values :** a map from column names to new column values
    - **whereClause :** the optional WHERE clause to apply when updating
    - **whereArgs :** an array of the value to apply to the WHERE clause
  - Return the number of rows affected

SQLiteDatabase

- int update (String table, ContentValues values, String whereClause, String[] whereArgs) :
  - Example :

```
SQLiteDatabase db = ...

ContentValues values = new ContentValues();
values.put("name", ”John");
String[] whereArgs = { "1" };

db.update(“persons”, values, "id=?", whereArgs);
```

SQLiteDatabase

- **int delete (String table, String whereClause, String[] whereArgs) :**
  - Convenience method for deleting rows in the Database
  - Three parameters :
    - **table :** the table to delete from
    - **whereClause :** the optional WHERE clause to apply when deleting
    - **whereArgs :** an array of the value to apply to the WHERE clause
  - Return the number of rows affected

SQLiteDatabase

- **int delete (String table, String whereClause, String[] whereArgs) :**
  - Example :

```
SQLiteDatabase db = ...

String[] whereArgs = { "1" };
db.delete("persons", "id=?", whereArgs);
```

SQLiteDatabase

- **Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy) :**
  - Query the given table, returning a Cursor over the result set
  - Seven parameters :
    - **table :** The table name to compile the query
    - **columns :** A list of which columns to return
    - **selection :** A filter declaring which rows to return, formatted as an SQL WHERE clause

SQLiteDatabase

- **Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy) :**
  - Seven parameters :
    - **selectionArgs :** You may include ?s in selection, which will be replaced by the values from selectionArgs
    - **groupBy :** A filter declaring how to group rows, formatted as an SQL GROUP BY clause
    - **having :** A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause
    - **orderBy :** How to order the rows, formatted as an SQL ORDER BY clause

SQLiteDatabase

- Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy) :
  - Example :

```
SQLiteDatabase db = ...

String[] columns = { ID_COLUMN, NAME_COLUMN };
String[] params = { "Cartman" };
Cursor result = db.query(TABLE_NAME, columns, "name=?",

      params, null, null, null, "1");
```

Cursor

- Provide access to the result set returned by a database query
- Methods commonly used are :
  - **getCount()** : returns the number of rows
  - **moveToFirst()** : moves the cursor to the first row
  - **moveToNext()** : moves the cursor to the next line
  - **isAfterLast()** : returns true if the cursor position is after the last row
  - **getColumnNames()** : returns a string array holding the names of all of the columns in the result set
  - **getColumnIndex(String name)** : return the index of the corresponding column name

Cursor

- Example of use :

```
String[] columns = { "id", "name"};

Cursor result = db.query("persons", columns, null,

null, null, null, null);
List<Person> persons = new ArrayList<Person>();

result.moveToFirst();
while(!result.isAfterLast()) {
   Person person = new Person();
   person.setId(result.getLong(0));
   person.setName(result.getString(1));
   persons.add(person);

   result.moveToNext();
}
result.close();

return persons;
```