Android: Data Storage

https://developer.android.com/guide/topics/data/data-storage.html

https://developer.android.com/training/data-storage/index.html

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2017-2018)

Outline

1. Overview
2. Shared Preferences
3. Internal Storage or External Storage
4. SQLite Databases
5. An SQLite Example: PDM18kotlin3v2

Outline

1. Overview
2. Shared Preferences
3. Internal Storage or External Storage
4. SQLite Databases
5. An SQLite Example: PDM18kotlin3v2

Storage Options

- Most Android apps need to save data
  - Information about app's state
  - User settings
  - Large amounts of information (in files and databases)
- Five options (THIS BLOCK OF SLIDES IS ABOUT THE FIRST FOUR)

  Shared Preferences Store primitive data in key-value pairs (in a shared preferences file).

  Internal Storage Store (private) data on the device memory (in arbitrary files in Android's file system).

  External Storage Store (public) data on the shared external storage.

  SQLite Databases Store structured data in a private database.

  Network Connection (Services) Store data on the web with your own network server.

Android provides a way for you to expose even your private data to other applications — with a content provider. A content provider is an optional component that exposes read/write access to your application data, subject to whatever restrictions you want to impose. (DEVOTED BLOCK OF SLIDES)

Outline

1. Overview
2. Shared Preferences
3. Internal Storage or External Storage
4. SQLite Databases
5. An SQLite Example: PDM18kotlin3v2

Get a Handle to a SharedPreferences

The SharedPreferences class provides a general framework to save and retrieve persistent key-value pairs of primitive data types (booleans, floats, ints, longs, and strings)—these data will persist across user sessions.

You can create a new shared preference file or access an existing one by calling one of two methods:

- getSharedPreferences()—Use this if you need multiple shared preference files identified by name, which you specify with the first parameter. You can call this from any Context in your app.
- getPreferences()—Use this from an Activity if you need to use only one shared preference file for the activity. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

Example

The following code is executed inside a Fragment. It accesses the shared preferences file that's identified by the resource string R.string.preference_file_key and opens it using the private mode so the file is accessible by only your app.

```
1        val sharedPref = activity?.getSharedPreferences(
2                getString(R.string.preference_file_key), Context.MODE_PRIVATE)
```

When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE_FILE_KEY"
Alternatively, if you need just one shared preference file for your activity, you can use the getPreferences() method:

```
1        val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE)
```

Caution: If you create a shared preferences file with MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE, then any other apps that know the file identifier can access your data.

- Deprecated in API level 17
- Starting from API level 24 attempting to use this modes throws a SecurityException.

Write-to and Read-from Shared Preferences

To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences.

- Pass the keys and values you want to write with methods such as putInt() and putString(). Then call commit() to save the changes.

Example

```
1        val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
2        with (sharedPref.edit()) {
3            putInt(getString(R.string.saved_high_score_key), newHighScore)
4            commit()
5        }
```

To retrieve values from a shared preferences file, call methods such as getInt() and getString().

- Provide the key for the value you want, and optionally a default value to return if the key isn't present.

Example

```
1        val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
2        val defaultValue = resources.getInteger(R.integer.saved_high_score_default_key)
3        val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key), defaultValue)
```

Outline

1 Overview

2 Shared Preferences

3 Internal Storage or External Storage

4 SQLite Databases

5 An SQLite Example: PDM18kotlin3v2

File system

- Internal storage *(always available)*
  - Files saved here accessible by only your app
  - System removes all app's files when app unistalled

  Best when you want to be sure that neither the user nor other apps can access your files.

- External storage *(not always available)*
  - Files saved here may be read outside of our control
  - When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from context.getExternalFilesDir().
  - Best place for files that don't require access restrictions
    - ⋆ Files we want to share with other apps
    - ⋆ Files the user can access with a computer
  - Permissions in the manifest needed
  - Before you do any work with the external storage you should always check whether the media is available

Save a File on Internal Storage

Acquire the appropriate directory as a File by calling one of two methods:

- context.filesDir. Returns a File representing an internal directory for your app.
- context.cacheDir. Returns a File representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the File() constructor, passing the File provided by one of the above methods that specifies your internal storage directory.

Example

```
1 val file = File(context?.filesDir, filename)
```

Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory.

Example (How to write some text to a file)

```
1         val filename = "myfile"
2         val string = "Hello world!"
3
4         val outputStream:FileOutputStream = applicationContext.openFileOutput(filename, Context.MODE_PRIVATE).apply {
5             write(string.toByteArray())
6             close()
7         }
```

Or, if you need to cache some files, you should instead use createTempFile().

Example

The following method extracts the file name from a URL and creates a file with that name in your app's internal cache directory:

```
1    fun getTempFile(context: Context, url: String):File {
2        val fileName = Uri.parse(url).lastPathSegment
3        return File.createTempFile(fileName, null, context.cacheDir)
4    }
```

Example: PDM17app3v1bis

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v1bis.git]

This example is based on PDM18kotlin3v1 and evolves the project by moving the information of the courses by the code in several XML files.
The difference can be found in the private constructor of **Model.kt**:

- Variante A
  - We use the resource courses_data.xml file in XML directory as a source of information on courses
- Variante B
  - We use the resource courses_data_bis.xml file in RAW directory as a source of information on courses
- Variante C
  - We use a local file as source of information on courses (and if not exist, it is created)
  - The information contained into the file are in 'right' XML format

Example: PDM18kotlin3v1bis

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v1bis.git]

Example (Variante C)

Model.kt: The following code inizialize an XML Parser for read the data from file:

```
1        val xmlFactoryObject = XmlPullParserFactory.newInstance()
2        val myParser = xmlFactoryObject.newPullParser()
3
4        val file = File(FileHelper.dataFileName)
5        if (!file.isFile) {
6            FileHelper.init()
7        }
8        val fileInputStream = MainActivity.applicationContext().openFileInput(FileHelper.dataFileName)
9        myParser.setInput(fileInputStream, null)
```

FileHelper.kt: This code write to the same file the string rappresentation of courses data:

```
1        val fos = MainActivity.applicationContext().openFileOutput(dataFileName, Context.MODE_PRIVATE)
2        fos.write(writer.toString().toByteArray())
3        fos.close()
```

Take a look at the emulator directory content (as root): /data/data/it.unito.di.educ.pdm18kotlin3v1bis/files

Further (asset) files

- /res/raw
  - To open these resources with a raw InputStream, use *resources?.openRawResource()* with the resource ID (R.raw.courses_data_bis)
  - To access original file names and file hierarchy, you should save some resources in the assets/ directory (instead of res/raw/)
    - ⋆ Files in assets/ are not given a resource ID, so you can read them only using AssetManager
- /res/xml
  - Arbitrary XML files that can be read at runtime by calling *resources?.getXml()*
  - Various XML configuration files must be saved here

Example: PDM18kotlin3v1bis

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v1bis.git]

Example (Variante A)

Model.kt: The following code inizialize an XML Parser for read the data from XML directory:

```
1    ...
2    val myParser = MainActivity.applicationContext().resources?.getXml(R.xml.courses_data) ?: return localCourses
3    ...
```

Example (Variante B)

Model.kt: The following code inizialize an XML Parser for read the data from RAW directory:

```
1    ...
2    val xmlFactoryObject = XmlPullParserFactory.newInstance()
3    val myParser = xmlFactoryObject.newPullParser()
4    myParser.setInput(MainActivity.applicationContext().resources?.openRawResource(R.raw.courses_data_bis), null);
5    ...
```

Save a File on External Storage

Although the external storage is modifiable by the user and other apps, there are two categories of files you might save here:

- Public files
  - Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
  - For example, photos captured by your app or other downloaded files.
- "Private" files
  - Files that rightfully belong to your app and should be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app.
  - For example, additional resources downloaded by your app or temporary media files.

Outline

1. Overview
2. Shared Preferences
3. Internal Storage or External Storage
4. SQLite Databases
5. An SQLite Example: PDM18kotlin3v2

SQLite

- Android provides full support for SQLite databases
  - Lightweight database based on SQL
  - Standard SQL syntax
- Any database is accessible by name to any class in the application, but not outside the application
- To create a new SQLite database, we must subclass SQLiteOpenHelper and override method onCreate() to execute SQLite commands to create tables

Define a Schema and Contract

SQL database schema: a formal declaration of how the database is organized.
Create a companion class (known as a contract class) which explicitly specifies the layout of your schema in a systematic and self-documenting way.

- A contract class is a container for constants that define names for URIs, tables, and columns.
- It allows you to use the same constants across all the other classes in the same package.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

Example

This snippet defines the table name and column names for a single table:

```
1  object FeedReaderContract {
2      // Table contents are grouped together in an anonymous object.
3      object FeedEntry : BaseColumns {
4          const val TABLE_NAME = "entry"
5          const val COLUMN_NAME_TITLE = "title"
6          const val COLUMN_NAME_SUBTITLE = "subtitle"
7  //-------------------------------------------------------------------------------------
8          ...
9  //-------------------------------------------------------------------------------------
10     }
11 }
```

Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables.

Example

Here we complete the inner class with some typical statements that create and delete a table:

```
object FeedReaderContract {
    // Table contents are grouped together in an anonymous object.
    object FeedEntry : BaseColumns {
        const val TABLE_NAME = "entry"
        const val COLUMN_NAME_TITLE = "title"
        const val COLUMN_NAME_SUBTITLE = "subtitle"
//-------------------------------------------------------------------------------------
        private const val SQL_CREATE_ENTRIES =
                "CREATE TABLE ${FeedEntry.TABLE_NAME} (" +
                        "${BaseColumns._ID} INTEGER PRIMARY KEY," +
                        "${FeedEntry.COLUMN_NAME_TITLE} TEXT," +
                        "${FeedEntry.COLUMN_NAME_SUBTITLE} TEXT)"

        private const val SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS ${FeedEntry.TABLE_NAME}"
//-------------------------------------------------------------------------------------
    }
}
```

A useful set of APIs is available in the SQLiteOpenHelper class.
Methods getWritableDatabase() and getReadableDatabase() allow us to write to and read from the database respectively—they can be long-running, be sure that you call them in a background thread!

- They return a SQLiteDatabase object that represents the database and provides methods for SQLite operations

query() allows one to issue queries to the database

- Every query returns a Cursor that points to all the rows and columns found by the query

Example

Here's an implementation of SQLiteOpenHelper that uses some of the commands shown above:

```
class FeedReaderDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(SQL_CREATE_ENTRIES)
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES)
        onCreate(db)
    }
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        onUpgrade(db, oldVersion, newVersion)
    }
    companion object {
        // If you change the database schema, you must increment the database version.
        const val DATABASE_VERSION = 1
        const val DATABASE_NAME = "FeedReader.db"
    }
}
```

Access a Database

To access your database, instantiate your subclass of SQLiteOpenHelper:

Example

```
1 val dbHelper = FeedReaderDbHelper(context)
```

Put Information into a Database

Insert data into the database by passing a ContentValues object to the insert() method:

Example

```
// Gets the data repository in write mode
val db = dbHelper.writableDatabase

// Create a new map of values, where column names are the keys
val values = ContentValues().apply {
    put(FeedEntry.COLUMN_NAME_TITLE, title)
    put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle)
}

// Insert the new row, returning the primary key value of the new row
val newRowId = db?.insert(FeedEntry.TABLE_NAME, null, values)
```

- The first argument for insert() is simply the table name.
- The second argument provides the name of a column in which the framework can insert NULL in the event that the ContentValues is empty (if you instead set this to "null", then the framework will not insert a row when there are no values).

Read Information from a Database

To read from a database, use the query() method, passing it your selection criteria and desired columns.

- The column list defines the data you want to fetch, rather than the data to insert.
- The results of the query are returned to you in a Cursor object.

Example

```
val db = dbHelper.readableDatabase

// Define a projection that specifies which columns from the database
// you will actually use after this query.
val projection = arrayOf(BaseColumns._ID, FeedEntry.COLUMN_NAME_TITLE, FeedEntry.COLUMN_NAME_SUBTITLE)

// Filter results WHERE "title" = 'My Title'
val selection = "${FeedEntry.COLUMN_NAME_TITLE} = ?"
val selectionArgs = arrayOf("My Title")

// How you want the results sorted in the resulting Cursor
val sortOrder = "${FeedEntry.COLUMN_NAME_SUBTITLE} DESC"

val cursor = db.query(
        FeedEntry.TABLE_NAME,   // The table to query
        projection,             // The array of columns to return (pass null to get all)
        selection,              // The columns for the WHERE clause
        selectionArgs,          // The values for the WHERE clause
        null,                   // don't group the rows
        null,                   // don't filter by row groups
        sortOrder               // The sort order
)
```

To look at a row in the cursor, use one of the Cursor move methods, which you must always call before you begin reading values.

- Generally, you should start by calling moveToFirst(), which places the "read position" on the first entry in the results.
- For each row, you can read a column's value by calling one of the Cursor get methods, such as getString() or getLong().
- For each of the get methods, you must pass the index position of the column you desire, which you can get by calling getColumnIndex() or getColumnIndexOrThrow().

Example

```
1 val itemIds = mutableListOf<Long>()
2 with(cursor) {
3     moveToFirst()
4     while (moveToNext()) {
5         val itemId = getLong(getColumnIndexOrThrow(BaseColumns._ID))
6         itemIds.add(itemId)
7     }
8 }
```

Delete Information from a Database

To delete rows from a table, provide selection criteria that identify the rows

- Then call delete()

Example

```
1 // Define 'where' part of query.
2 val selection = "${FeedEntry.COLUMN_NAME_TITLE} LIKE ?"
3 // Specify arguments in placeholder order.
4 val selectionArgs = arrayOf("MyTitle")
5 // Issue SQL statement.
6 val deletedRows = db.delete(FeedEntry.TABLE_NAME, selection, selectionArgs)
```

Update a Database

To modify a subset of your database values, use update()

- Updating the table combines the content values syntax of insert() with the where syntax of delete().

Example

```
val db = dbHelper.writableDatabase

// New value for one column
val title = "MyNewTitle"
val values = ContentValues().apply {
    put(FeedEntry.COLUMN_NAME_TITLE, title)
}

// Which row to update, based on the title
val selection = "${FeedEntry.COLUMN_NAME_TITLE} LIKE ?"
val selectionArgs = arrayOf("MyOldTitle")
val count = db.update(
        FeedEntry.TABLE_NAME,
        values,
        selection,
        selectionArgs)
```

Outline

1. Overview
2. Shared Preferences
3. Internal Storage or External Storage
4. SQLite Databases
5. An SQLite Example: PDM18kotlin3v2

An SQLite example: PDM17app3v2

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v2.git]

This project is an update of PDM18kotlin3v1,

the files added and modified in this example are:

- Model.java
  - Removed the internal variable with the courses list
  - method getSize: use the class DatabaseUtils to get the size of a table
  - method getCourseElement: performs an raw SQL query to get the course information
  - method getCourseDetail: performs a query to get the details of a course
- CourseDBHelper.java
  - Define two internal object, one for each table created into the DB:
    - CourseTable
    - CourseDetailTable
  - Override the methods for creating and upgrading the DB

Model.kt (1/2)

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v2.git]

Raw SQL query:

```
1   fun getCourseElement(position:Int):Course {
2       val db = dbHelper.readableDatabase
3
4       val sql = "SELECT " + CoursesDBHelper.CourseTable.COLUMN_NAME + " ," +
5               CoursesDBHelper.CourseTable.COLUMN_CODICE + " " +
6               "FROM " + CoursesDBHelper.CourseTable.TABLE_NAME + " " +
7               "ORDER BY " + CoursesDBHelper.CourseTable.COLUMN_NAME + " ASC " +
8               "LIMIT " + position + ",1"
9       val cursor = db.rawQuery(sql, null)
10
11      val course = Course("Undefined","empty","","")
12      if (cursor.moveToNext()) {
13          course.nome=cursor.getString(0)
14          course.codice=cursor.getString(1)
15      }
16      cursor.close()
17
18      return course
19  }
```

Model.java (2/2)

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v2.git]

Cursor query:

```
1     fun getCourseDetail(code:String): Course {
2         val db = dbHelper.readableDatabase
3 /*      db.query(
4                 String table, String[] columns,
5                 String selection, String[] selectionArgs,
6                 String groupBy, String having, String orderBy)  */
7         val cursor = db.query(
8             CoursesDBHelper.CourseDetailTable.TABLE_NAME,
9             arrayOf(
10                CoursesDBHelper.CourseDetailTable.COLUMN_NAME,       CoursesDBHelper.CourseDetailTable.COLUMN_CODICE,
11                CoursesDBHelper.CourseDetailTable.COLUMN_MATERIALE,  CoursesDBHelper.CourseDetailTable.COLUMN_OBIETTIVI
12            ),
13            "${CoursesDBHelper.CourseDetailTable.COLUMN_CODICE}=?",
14            arrayOf(code),
15            "",  "",  ""
16        )
17        ...
18        if (cursor.moveToNext()) {
19            course.nome=cursor.getString(0)
20            ...
21        }
22        cursor.close()
23        return course
24    }
```

CoursesDBHelper.java (1/2)

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v2.git]

```
1  class CoursesDBHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
2
3      companion object {
4          // Su file system: /data/data/it.unito.di.educ.pdm17app3v2/databases
5          private const val DATABASE_NAME = "Didattica"
6          private const val DATABASE_VERSION = 1
7      }
8
9      object CourseTable {
10         const val TABLE_NAME = "corsi"
11
12         const val COLUMN_NAME = "name"
13         const val COLUMN_CODICE = "codice"
14         const val COLUMN_TIMESTAMP = "timestamp"
15
16         internal const val SQL_CREATE_TABLE = "CREATE TABLE $TABLE_NAME (" +
17                 "$COLUMN_NAME TEXT NOT NULL, " +
18                 "$COLUMN_CODICE TEXT NOT NULL, " +
19                 "$COLUMN_TIMESTAMP DATETIME DEFAULT CURRENT_TIMESTAMP," +
20                 "CONSTRAINT ${TABLE_NAME}_pk PRIMARY KEY ($COLUMN_CODICE) )"
21
22         internal const val SQL_DROP_TABLE = "DROP TABLE IF EXISTS $TABLE_NAME"
23     }
24     ...
25 }
```

CoursesDBHelper.java (2/2)

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin3v2.git]

```
1   override fun onCreate(db: SQLiteDatabase) {
2       Log.d("PDM18kotlin3", "Creazione delle tabelle del DB e inizializzazione")
3       db.execSQL(CourseTable.SQL_CREATE_TABLE)
4       db.execSQL(CourseDetailTable.SQL_CREATE_TABLE)
5
6       val row = ContentValues()
7       row.clear()
8       row.put(CourseDetailTable.COLUMN_CODICE, "MFN1348")
9       row.put(CourseDetailTable.COLUMN_NAME, "Agenti Intelligenti")
10      db.insert(CourseTable.TABLE_NAME, null, row)
11
12      row.put(CourseDetailTable.COLUMN_MATERIALE, "..." )
13      row.put(CourseDetailTable.COLUMN_OBIETTIVI, "..." )
14      db.insert(CourseDetailTable.TABLE_NAME, null, row)
15      ...
16  }
```

```
1   override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
2       db.execSQL(CourseTable.SQL_DROP_TABLE)
3       db.execSQL(CourseDetailTable.SQL_DROP_TABLE)
4       onCreate(db)
5   }
```