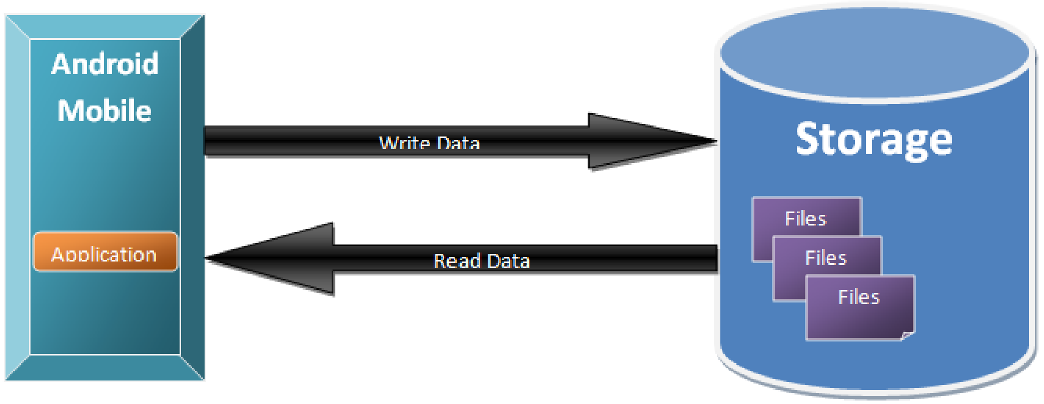
Data Persistence in Android

File IO

Text file

- internal storage

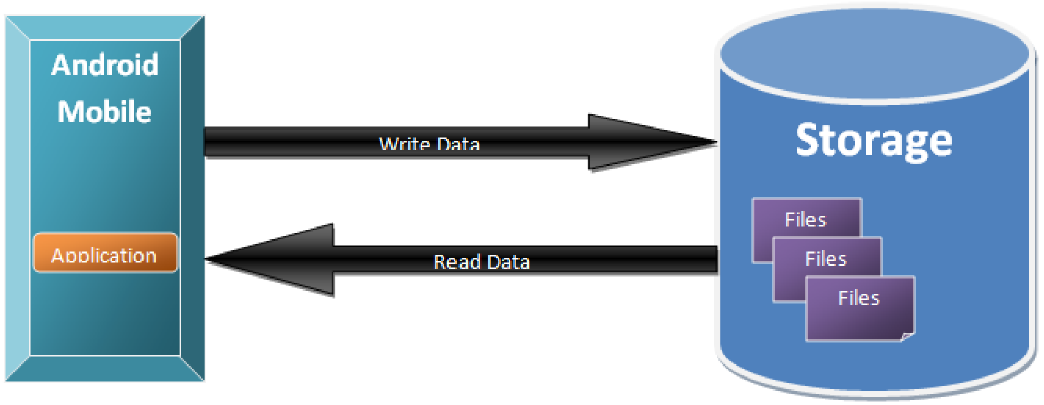


```
// use internal storage directory (you don't need permission)
// path = "/data/user/0/<package>/files"
val path = this.getFilesDir()
// Create your directory:
val directory = File(path, "tmp")
directory.mkdirs()

// Then create your file:
val file = File(directory, "test.txt")

// Then you can write to it:
file.writeText("Prima riga\n")
file.appendText("aggiungo una nuova riga!\n")
```

```
val inputAsString = FileInputStream(file).bufferedReader().use { it.readText() }
Log.i("MainActivity", inputAsString)
```

Text file

- External storage



```
// use external storage directory (you need permission)
// /storage/emulated/0/Android/data/<your package>/files
val path = this.getExternalFilesDir(null)

// Create your directory:
val directory = File(path, "tmp")
directory.mkdirs()

// Then create your file:
val file = File(directory, "test.txt")

// Then you can write to it:
file.writeText("Prima riga\n")
file.appendText("aggiungo una nuova riga!\n")
```

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

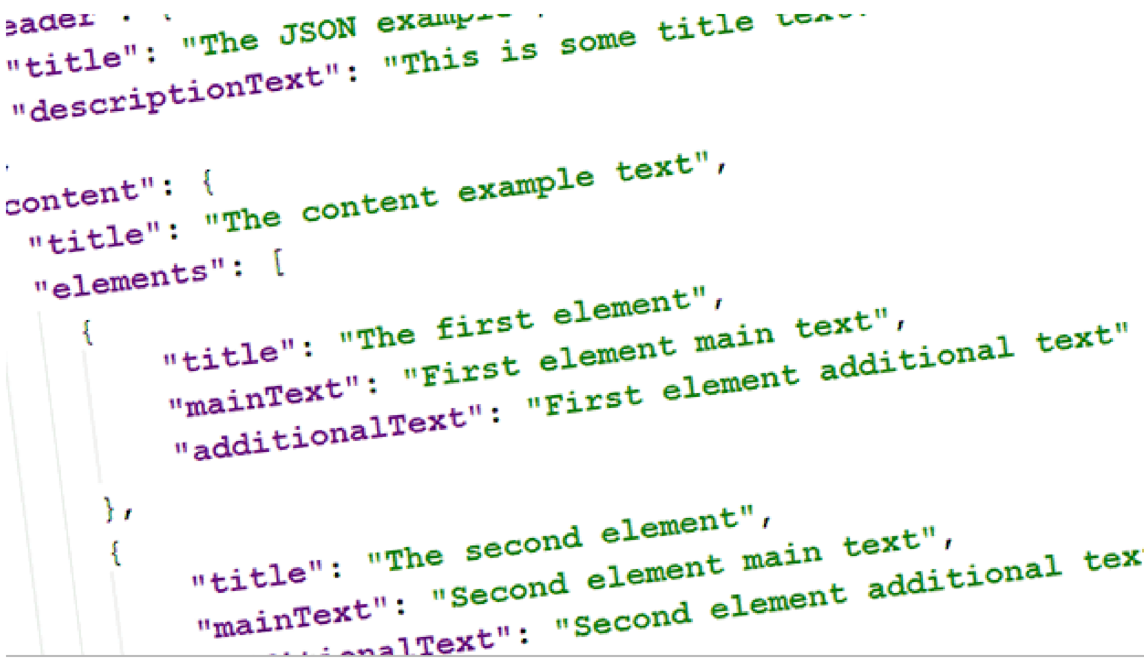
```
val inputAsString = FileInputStream(file).bufferedReader().use { it.readText() }
Log.i("MainActivity", inputAsString)
```

```
{
    "book1": {
        "name": "high school mathematics",
        "price": 12
    },
    "book2": {
        "name": "advanced high school mathematics",
        "price": 14
    }
}
```

JSON file

JSON

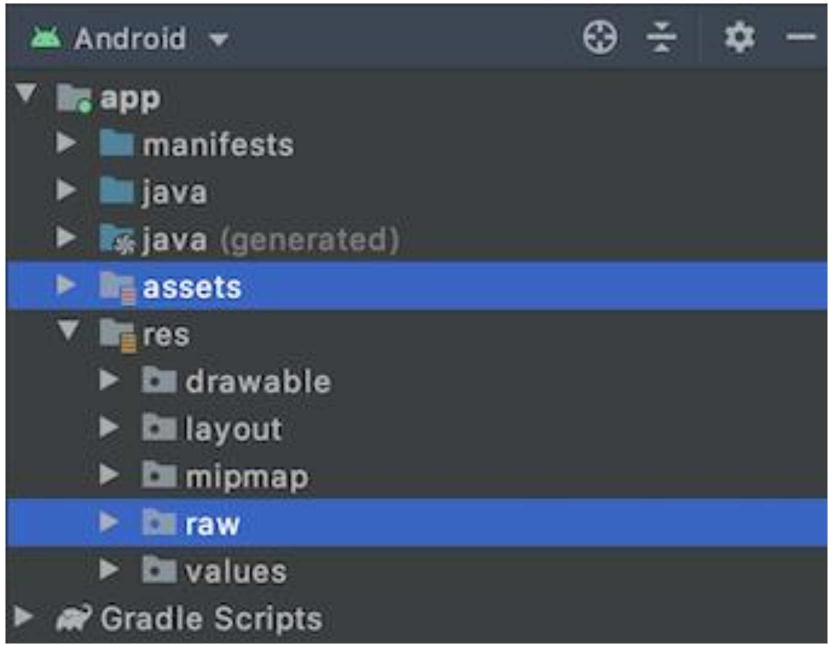
- stores simple data structures and objects in JavaScript Object Notation (JSON) format,
- It is a standard data interchange format.
- It is primarily used for transmitting data between a web application and a server.
- JSON files are lightweight, text-based, human-readable, and can be edited using a text editor.



```
eader": {
"title": "The JSON example",
"descriptionText": "This is some title text"
,
content": {
 "title": "The content example text",
 "elements": [
   {
       "title": "The first element",
       "mainText": "First element main text",
       "additionalText": "First element additional text"

   },
   {
       "title": "The second element",
       "mainText": "Second element main text",
       onalText": "Second element additional tex
```

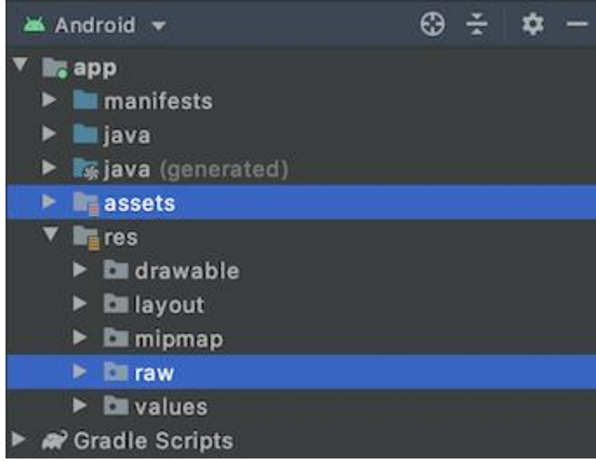
Assets Folder

- provides a way to add arbitrary files like text, XML, HTML, fonts, music, and video in the application.
- If one tries to add these files as “resources“, Android will treat them into its resource system and you will be unable to get the raw data.
- If one wants to access data untouched, Assets are one way to do it



We can do the same things by creating a **Resource** Raw Folder.

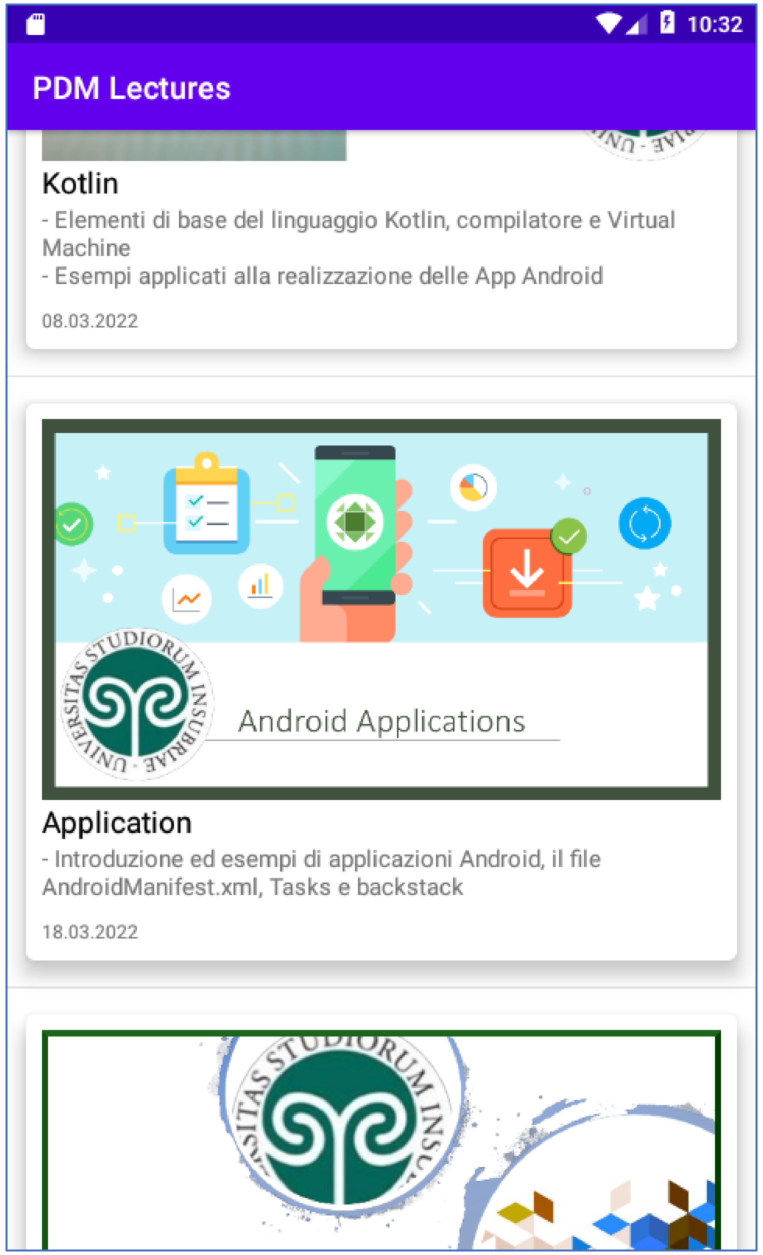
Why do we need to write in the asset folder?



Assets or res/raw Folder?

- **1. Flexible File Name: (assets is better)**
  - **assets:** The developer can name the file name in any way, like having capital letters (fileName) or having space (file name).
  - **res/raw:** In this case, the name of the file is restricted. File-based resource names must contain only lowercase **a-z, 0-9, or underscore.**
- **2. Store in subdirectory: (possible in assets)**
  - **assets:** If the developer wants to categories the files into subfolders, then he/she can do it in assets like below.
  - **res/raw:** In this case, files can only be in the root folder.

Custom Adapter Example using a JSON file


10:32

PDM Lectures

Kotlin

- Elementi di base del linguaggio Kotlin, compilatore e Virtual Machine
- Esempi applicati alla realizzazione delle App Android

08.03.2022

Application

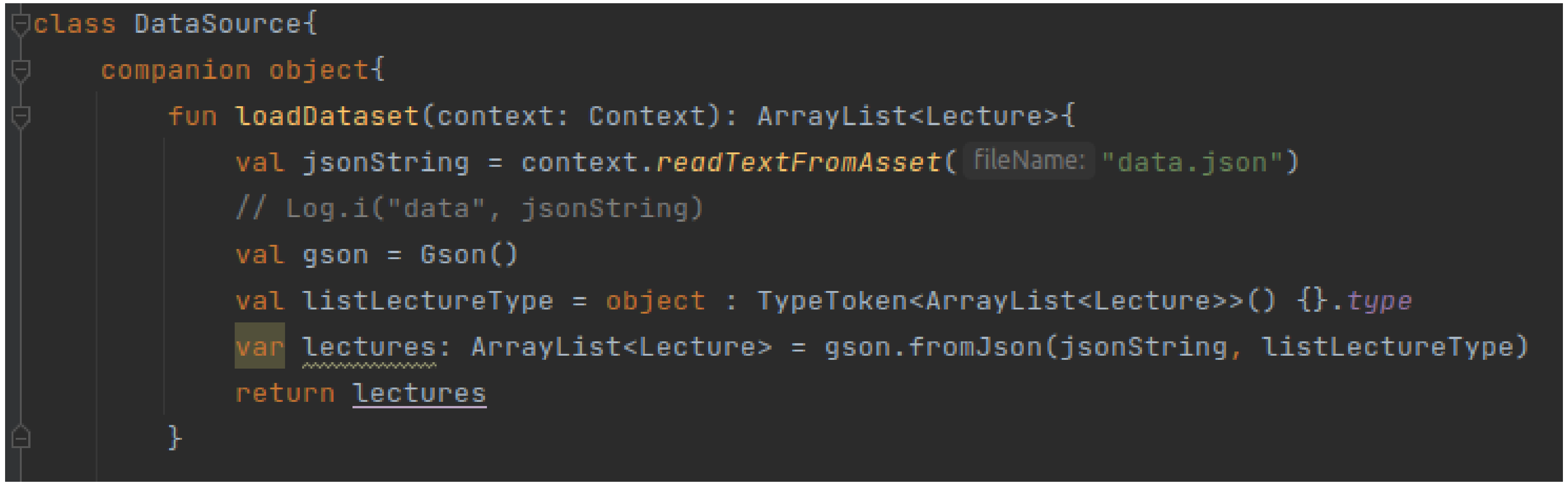
- Introduzione ed esempi di applicazioni Android, il file AndroidManifest.xml, Tasks e backstack

18.03.2022

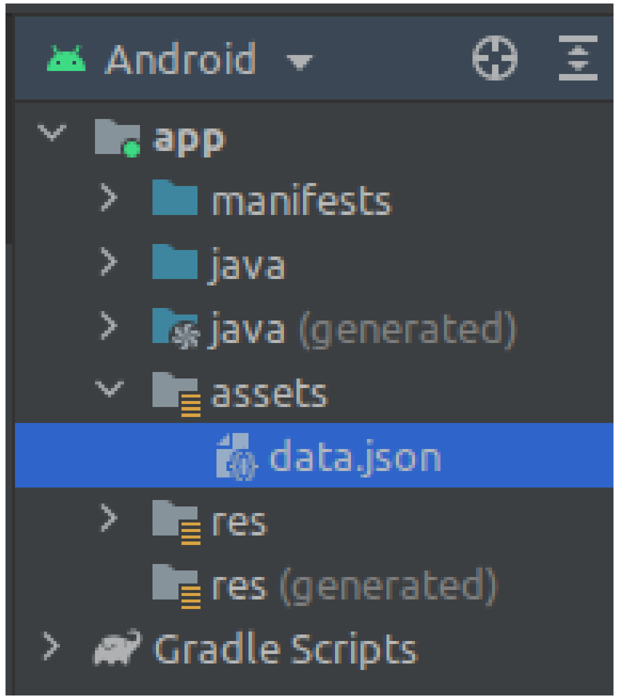
```
dependencies {
  implementation 'com.google.code.gson:gson:2.8.6'
```

Add data.json into Assets

- **1. Download the file from**
  - https://github.com/ignaziogallo/PDM/blob/master/data.json
- **2. Store it in assets directory**
- **3. Load data from json file using GSon library**



```
class DataSource{
    companion object{
        fun loadDataset(context: Context): ArrayList<Lecture>{
            val jsonString = context.readTextFromAsset( fileName: "data.json")
            // Log.i("data", jsonString)
            val gson = Gson()
            val listLectureType = object : TypeToken<ArrayList<Lecture>>() {}.type
            var lectures: ArrayList<Lecture> = gson.fromJson(jsonString, listLectureType)
            return lectures
        }
```



Database

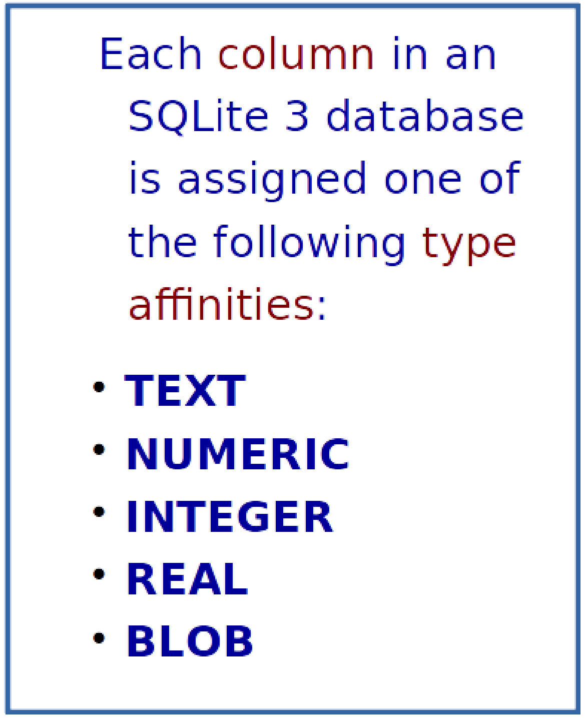
SQLite

SQLite

- SQLite is a relational database management system (RDBMS): open-source, standards-compliant, lightweight.
- It is implemented as a compact C library (rather than running as a **separate** ongoing **process**).
- An SQLite database is an integrated part of the application that created it (No separate server process).
  - this reduces external dependencies,
  - minimizes latency
  - simplifies transaction locking and synchronization.
- It is included as part of the Android software stack and exposed via a Java class.

SQLite data type

- SQLite supports the concept of type affinity on columns.
  - Any column can still store **any type of data** but the preferred storage class for a column is called its affinity.
- SQLite differs from many conventional database engines by loosely typing each column:
  - column values are not required to conform to a single type; instead, **each value is typed individually in each row**.
  - type checking isn't necessary when assigning or extracting values from each column within a row.


Each column in an SQLite 3 database is assigned one of the following type affinities:

- **TEXT**
- **NUMERIC**
- **INTEGER**
- **REAL**
- **BLOB**

SQLite data type

- SQLite's typing system in a column can hold a value of any type.
- So even with the following table definition:

```
CREATE TABLE number_values (
  value INTEGER NOT NULL
);
```

- We can run

```
INSERT INTO number_values VALUES ('foo')
```

and SQLite will happily store 'foo' into the value column.

Principal concepts

`android.database.sqlite.SQLiteOpenHelper`

- Helper (abstract) class used to implement the best practice pattern for **creating**, **opening**, and **upgrading** databases.

```
class DataBaseHelper(var context: Context) : SQLiteOpenHelper(context, DATABASENAME, null, DBVERSION){

  override fun onCreate(db: SQLiteDatabase?) {
      db?.execSQL("CREATE TABLE " + TABLENAME + " (  ... )" )
  }

  override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {
      db?.execSQL("DROP TABLE IF EXISTS " + TABLENAME)
      onCreate(db)
  }
```

Called when the database is created for the first time.

Called when the database needs to be upgraded.

How To: Android SQLite onUpgrade()

Principal concepts

`android.database.sqlite.SQLiteDatabase`

- The class exposing methods to manage a SQLite database.

```
class DataBaseHelper(var context: Context) : SQLiteOpenHelper(context, DATABASENAME, null, 1){

    ...
    val database = this.writableDatabase
    ...
    val result = database.insert(TABLENAME, null, contentValues)

    ...
    val db = this.readableDatabase
    val result = db.rawQuery("Select * from $TABLENAME", null)
```

WritableDatabase Create and/or open a database that will be used for reading and writing.

ReadableDatabase is faster. If you don't need to write anything, then readable database should be used.

Principal concepts

`android.database.sqlite.SQLiteCursor`

- Used to describe the result of database queries.
  - Cursors are pointers to the result set within the underlying data;
  - They provide **methods** to manage the result set.

```
int getCount()

int getPosition()

boolean isLast()

String[] getColumnNames()

float getFloat(int columnIndex)

double getDouble(int columnIndex)
```

SQLiteCursor

```
val result = db.rawQuery(query, null)
if (result.moveToFirst())
  do {
     ....
  } while (result.moveToNext())
```

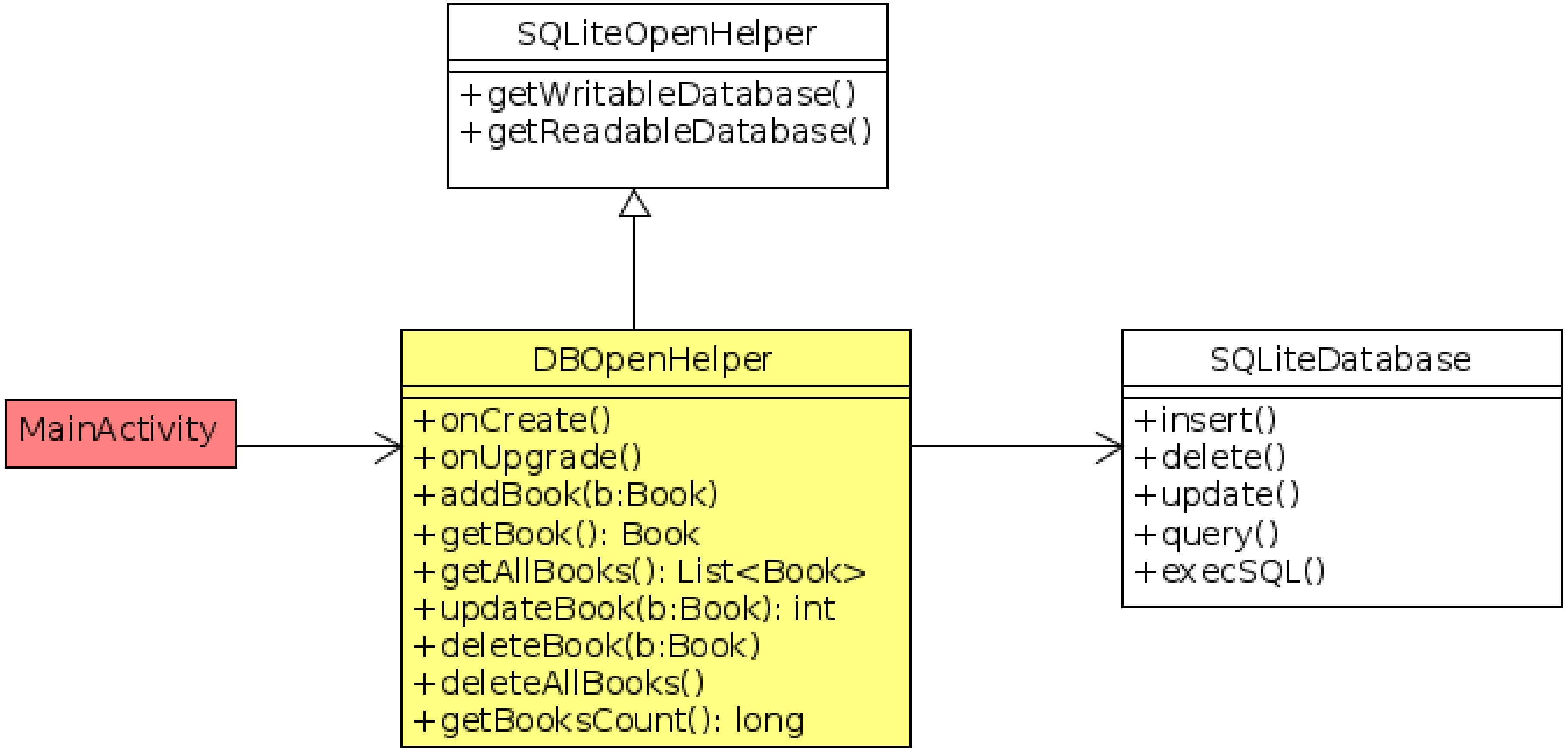
Principal concepts

`android.content.ContentValues`

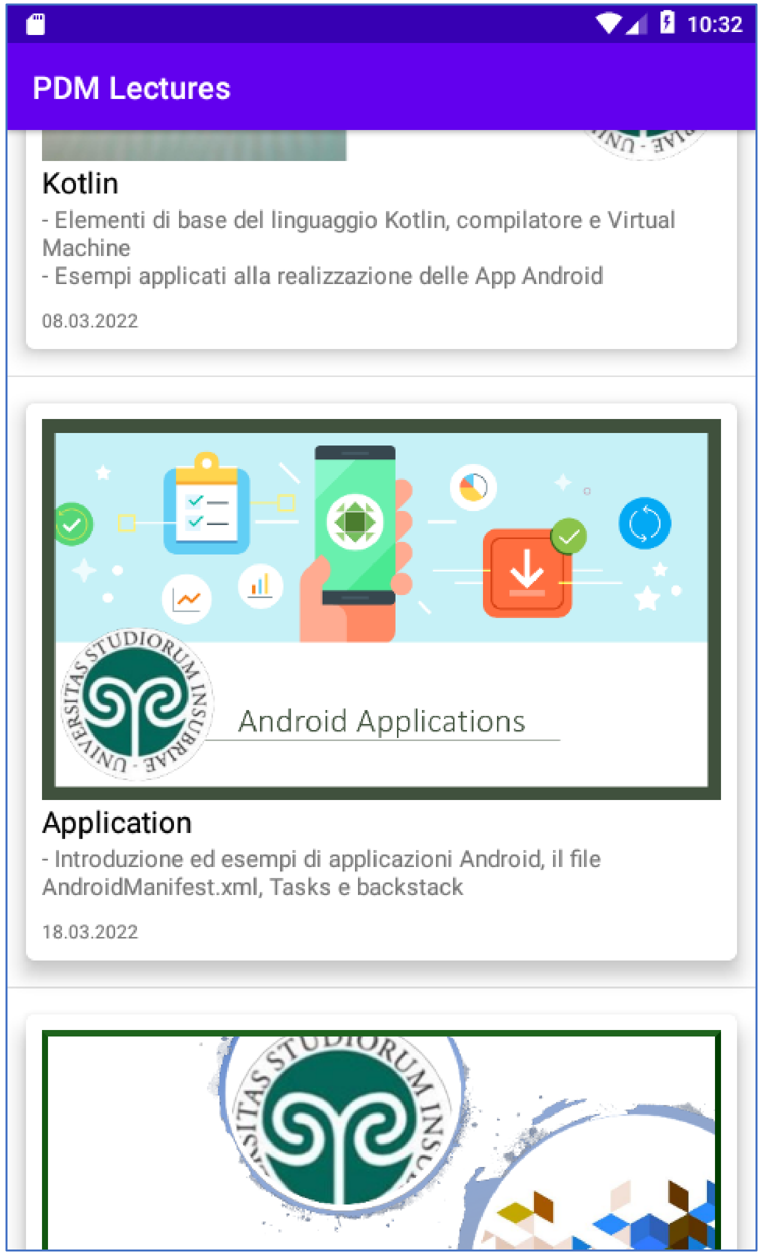
- Set of Name/Value pairs (NVP) used to insert rows into tables.

```
val contentValues = ContentValues()
contentValues.put(COL_NAME, user.name)
contentValues.put(COL_SURNAME, user.surname)

val result = database.insert(TABLENAME, null, contentValues)
```

Classes


SQLiteOpenHelper
+getWritableDatabase()
+getReadableDatabase()
MainActivity
DBOpenHelper
+onCreate()
+onUpgrade()
+addBook(b:Book)
+getBook(): Book
+getAllBooks(): List<Book>
+updateBook(b:Book): int
+deleteBook(b:Book)
+deleteAllBooks()
+getBooksCount(): long
SQLiteDatabase
+insert()
+delete()
+update()
+query()
+execSQL()

Custom Adapter Example using a SQLite DB



Kotlin

- Elementi di base del linguaggio Kotlin, compilatore e Virtual Machine
- Esempi applicati alla realizzazione delle App Android

08.03.2022

Application

- Introduzione ed esempi di applicazioni Android, il file AndroidManifest.xml, Tasks e backstack

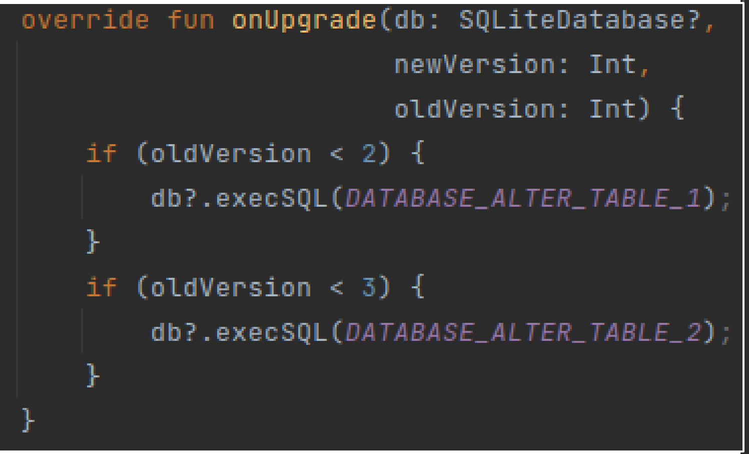
18.03.2022

MainActivity

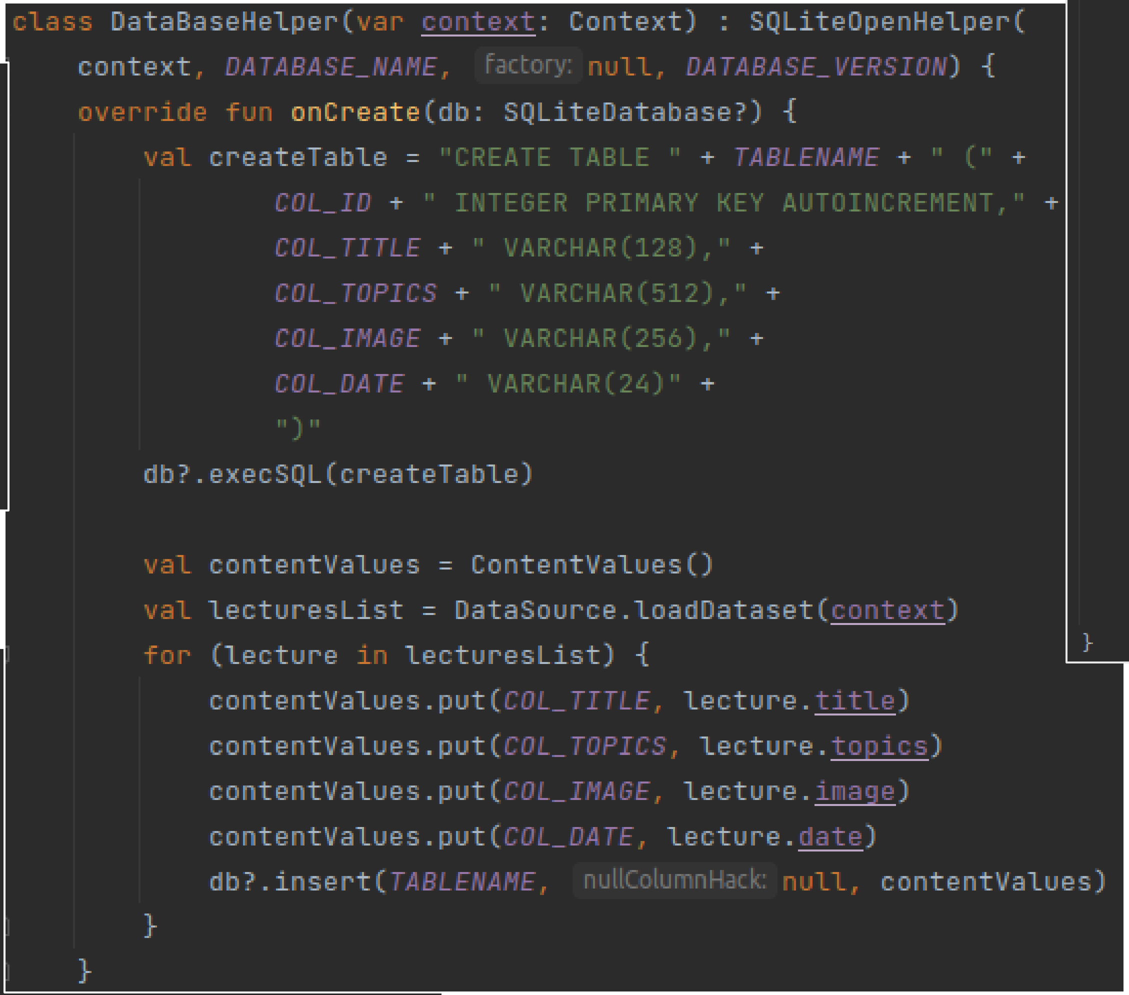
- Read data from SQLite database
- Pass data to the List Adapter for visualization

```
class MainActivity : AppCompatActivity() {
private lateinit var lectureAdapter: LectureListAdapter

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main_list)

  val db = DataBaseHelper(this)
  var lectures = db.readData()

  lectureAdapter = LectureListAdapter(this, lectures)
  list_view.adapter = lectureAdapter
  ...
```

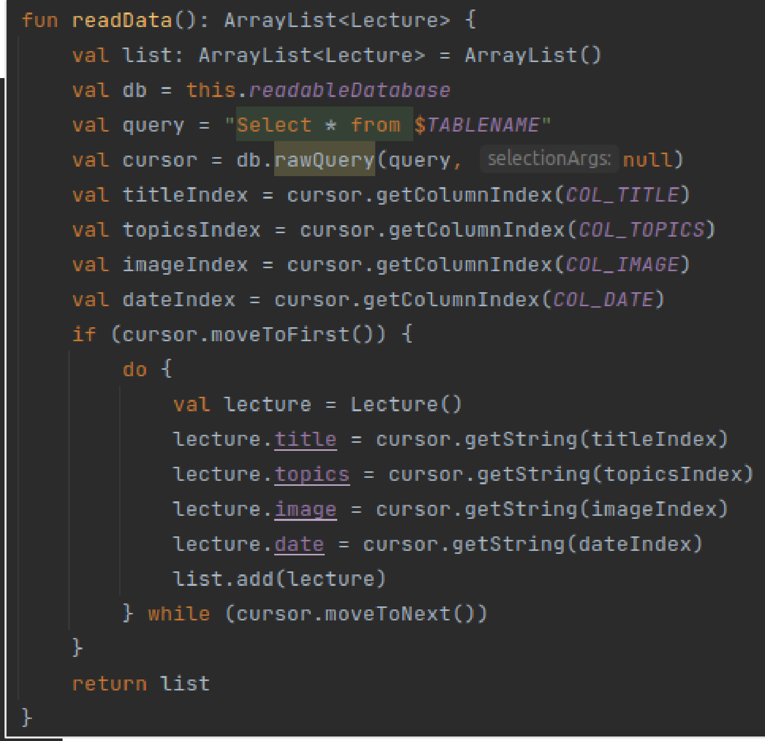
DataBaseHelper



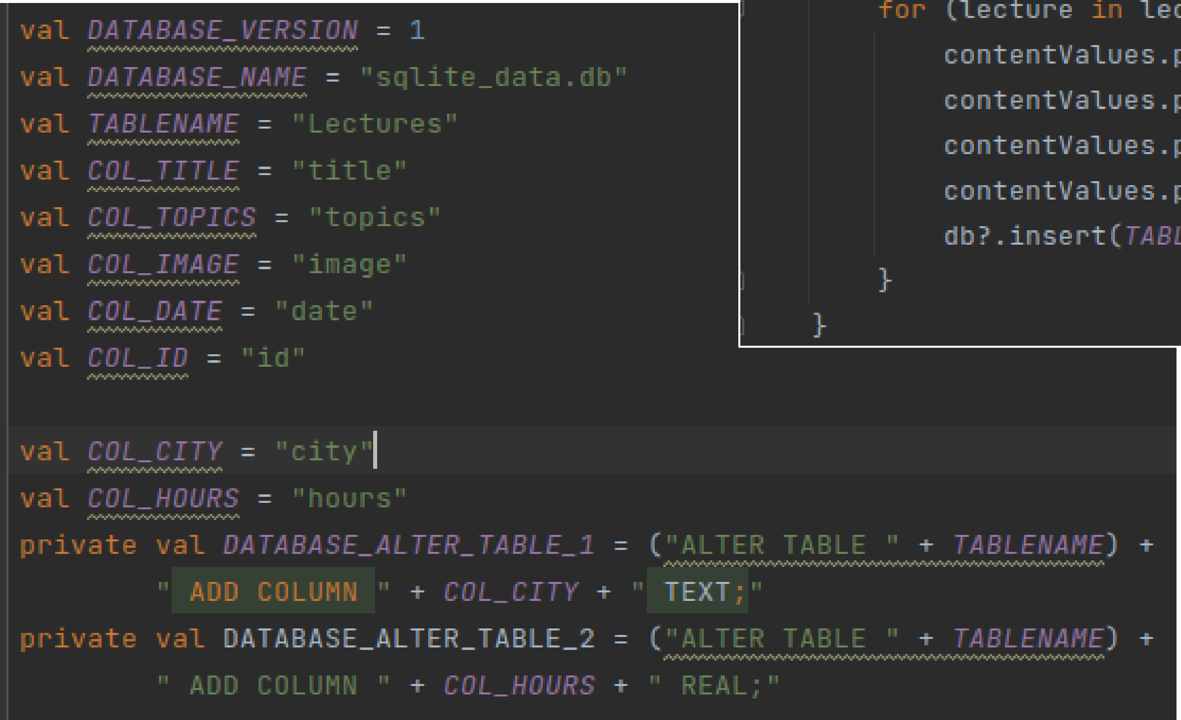
```
override fun onUpgrade(db: SQLiteDatabase?,
                       newVersion: Int,
                       oldVersion: Int) {
    if (oldVersion < 2) {
        db?.execSQL(DATABASE_ALTER_TABLE_1);
    }
    if (oldVersion < 3) {
        db?.execSQL(DATABASE_ALTER_TABLE_2);
    }
}
```



```
class DataBaseHelper(var context: Context) : SQLiteOpenHelper(
    context, DATABASE_NAME,  factory: null, DATABASE_VERSION) {
    override fun onCreate(db: SQLiteDatabase?) {
        val createTable = "CREATE TABLE " + TABLENAME + " (" +
                COL_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
                COL_TITLE + " VARCHAR(128)," +
                COL_TOPICS + " VARCHAR(512)," +
                COL_IMAGE + " VARCHAR(256)," +
                COL_DATE + " VARCHAR(24)" +
                ")"
        db?.execSQL(createTable)

        val contentValues = ContentValues()
        val lecturesList = DataSource.loadDataset(context)
        for (lecture in lecturesList) {
            contentValues.put(COL_TITLE, lecture.title)
            contentValues.put(COL_TOPICS, lecture.topics)
            contentValues.put(COL_IMAGE, lecture.image)
            contentValues.put(COL_DATE, lecture.date)
            db?.insert(TABLENAME,  nullColumnHack: null, contentValues)
        }
    }
```



```
fun readData(): ArrayList<Lecture> {
    val list: ArrayList<Lecture> = ArrayList()
    val db = this.readableDatabase
    val query = "Select * from $TABLENAME"
    val cursor = db.rawQuery(query,  selectionArgs: null)
    val titleIndex = cursor.getColumnIndex(COL_TITLE)
    val topicsIndex = cursor.getColumnIndex(COL_TOPICS)
    val imageIndex = cursor.getColumnIndex(COL_IMAGE)
    val dateIndex = cursor.getColumnIndex(COL_DATE)
    if (cursor.moveToFirst()) {
        do {
            val lecture = Lecture()
            lecture.title = cursor.getString(titleIndex)
            lecture.topics = cursor.getString(topicsIndex)
            lecture.image = cursor.getString(imageIndex)
            lecture.date = cursor.getString(dateIndex)
            list.add(lecture)
        } while (cursor.moveToNext())
    }
    return list
}
```



```
val DATABASE_VERSION = 1
val DATABASE_NAME = "sqlite_data.db"
val TABLENAME = "Lectures"
val COL_TITLE = "title"
val COL_TOPICS = "topics"
val COL_IMAGE = "image"
val COL_DATE = "date"
val COL_ID = "id"

val COL_CITY = "city"
val COL_HOURS = "hours"
private val DATABASE_ALTER_TABLE_1 = ("ALTER TABLE " + TABLENAME) +
        " ADD COLUMN " + COL_CITY + " TEXT;"
private val DATABASE_ALTER_TABLE_2 = ("ALTER TABLE " + TABLENAME) +
        " ADD COLUMN " + COL_HOURS + " REAL;"
```

Room Persistence Library

Room is part of the Android Jetpack

Data Binding
Lifecycles
LiveData
Navigation new!
Paging new!
Room
ViewModel
WorkManager new!
AppCompat
Android KTX new!
Multidex
Test
Architecture
UI
Android Jetpack
Foundation
Behavior
Animation & Transitions
Auto, TV & Wear
Emoji
Fragment
Layout
Palette
Download Manager
Media & Playback
Permissions
Notifications
Sharing
new! Slices

***Room** provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.*