Unit- 4

Data and Storage API

→ Android uses a file system that's similar to disk-based file systems on other platforms. Android provides several options for we to save persistent application data.

→ The solution we choose depends on our specific needs, such as whether the data should be private to our application or accessible to other applications (and the user) and how much space our data requires. The system provides several options for we to save our app data:

- Shared Preferences
  Store private primitive data in key-value pairs.
- Internal Storage
  Store private data on the device memory.
- External Storage
  Store public data on the shared external storage.
- SQLite Databases
  Store structured data in a private database.

❖ SharedPreferences:

→ Android Shared Preferences allow the activities or applications to store and retrieve data in the form of key and value. The data stored in the application remains to persist even if the app is closed until it has deleted or cleared.

→ The Android setting files use Shared Preferences to store the app setting data in the form of XML file under **data/data/{application package}/share_prefs** directory.

→ To access the Shared Preferences in our application, we need to get the instance of it using any of the following methods.

- getPreferences()
- getSharedPreferences()
- getDefaultSharedPreferences()

→ The first parameter is the key and the second parameter is the MODE. Apart from private there are other modes available that are listed below –

val sharedPreferences: SharedPreferences = this.getSharedPreferences(preferences_fileName: String ,mode: Int)

Here preferences_fileName is the Shared Preferences file name and mode is the operational mode of the file.

→ The modifications over the preferences data are performed through the **SharedPreferences.Editor** object.

→ val editor:SharedPreferences.Editor = sharedPreferences.edit()

→ To delete the preferences data of application we call the method:

- **editor.remove("key"):** it removes the specified key's value
- **editor.clear():** it removes all preferences data

→ The data stored in Shared preferences will lose when we perform any of the following operation:

- Uninstalling the application.
- Clearing the application data through setting.

→ Different Modes:

MODE_APPEND	This will append the new preferences with the already existing preferences

MODE_ENABLE_WRITE_AHEAD_LOGGING	Database open flag. When it is set , it would enable write ahead logging by default
MODE_MULTI_PROCESS	This method will check for modification of preferences even if the sharedpreference instance has already been loaded
MODE_PRIVATE	By setting this mode, the file can only be accessed using calling application
MODE_WORLD_READABLE	This mode allow other application to read the preferences
MODE_WORLD_WRITEABLE	This mode allow other application to write the preferences

→ **Methods of SharedPreferences class:**

abstract fun contains(key: String): Boolean	This method is used to check whether the preferences contain a preference.
abstract fun edit():SharedPreferences.Editor!	This method is used to create a new Editor for these preferences, through which we can make modifications to the data in the preferences and atomically commit those changes back to the SharedPreferences object.
abstract fun getAll(): MutableMap<String!, *>!	This method is used to retrieve all values from the preferences.
abstract fun getBoolean(key: String!, defValue: Boolean): Boolean	This method is used to retrieve a boolean value from the preferences.
abstract fun getFloat(key: String!, defValue: Boolean): Float	This method is used to retrieve a float value from the preferences.
abstract fun getInt(key: String!, defValue: Boolean): Int	This method is used to retrieve an int value from the preferences.
abstract fun getLong(key: String!, defValue: Boolean): Long	This method is used to retrieve a long value from the preferences.
abstract fun getString(key: String!, defValue: String?): String?	This method is used to retrieve a String value from the preferences.
abstract fun registerOnSharedPreferenceChangeListener(listener: SharedPreferences.OnSharedPreferenceChangeListener!): Unit	This method is used to registers a callback to be invoked when a change happens to a preference.
abstract fun unregisterOnSharedPreferenceChangeListener(listener: SharedPreferences.OnSharedPreferenceChangeListener!): Unit	This method is used to unregisters a previous callback.

→ Nested classes of Shared Preferences

- **SharedPreferences.Editor:** Interface used to write(edit) data in the SP file. Once editing has been done, one must commit() or apply() the changes made to the file.

- **SharedPreferences.OnSharedPreferenceChangeListener():** Called when a shared preference is changed, added, or removed. This may be called even if a preference is set to its existing value. This callback will be run on our main thread.

→ We can save something in the sharedpreferences by using SharedPreferences.Editor class. We will call the edit method of SharedPreference instance and will receive it in an editor object. Its syntax is –

```
Editor editor = sharedpreferences.edit();
editor.putString("key", "value");
editor.commit();
```

→ **Methods of Editor class:**

abstract apply(): Unit	It is an abstract method. It will commit our changes back from editor to the sharedPreference object we are calling
abstract fun clear(): SharedPreferences.Editor!	It will remove all values from the editor
abstract fun commit(): Boolean	Commit preferences changes back from this Editor to the SharedPreferences object it is editing.
fun remove(key: String!)	It will remove the value whose key has been passed as a parameter
abstract fun putBoolean(key: String!,value: Long): SharedPreferences.Editor!	It will save a Boolean value in a preference editor
abstract fun putLong(key: String!,value: Long): SharedPreferences.Editor!	It will save a long value in a preference editor
abstract fun putInt(key: String!,value: Long): SharedPreferences.Editor!	It will save a integer value in a preference editor
abstract fun putFloat(key: String!,value: Long): SharedPreferences.Editor!	It will save a float value in a preference editor
abstract fun putString(key: String!,value: Long): SharedPreferences.Editor!	It will save a string value in a preference editor

→ **Example:**
res/layout/activiy_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <TableLayout
      android:layout_width="300dp"
      android:layout_height="wrap_content"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
```

```
android:layout_marginEnd="8dp"
android:layout_marginBottom="8dp"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintHorizontal_bias="0.481"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
app:layout_constraintVertical_bias="0.038">


<TableRow
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="0"
      android:layout_marginStart="10sp"
      android:layout_marginLeft="10sp"
      android:text="Enter Id"
      android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />

   <EditText
      android:id="@+id/editId"
      android:layout_width="150dp"
      android:layout_height="wrap_content"
      android:layout_column="1"
      android:layout_marginStart="50sp"
      android:layout_marginLeft="50sp"
      android:hint="id"
      android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />
</TableRow>

<TableRow>

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="0"
      android:layout_marginStart="10sp"
      android:layout_marginLeft="10sp"
      android:text="Enter Name"
      android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />

   <EditText
      android:id="@+id/editName"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1"
      android:layout_marginStart="50sp"
      android:layout_marginLeft="50sp"
      android:hint="name"
      android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />
</TableRow>
```

```
    <TableRow android:layout_marginTop="60dp">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:layout_marginStart="10sp"
            android:layout_marginLeft="10sp"
            android:text="Your Id"
            android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />

        <TextView
            android:id="@+id/textViewShowId"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:layout_marginStart="50sp"
            android:layout_marginLeft="50sp"
            android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />
    </TableRow>

    <TableRow android:layout_marginTop="20dp">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:layout_marginStart="10sp"
            android:layout_marginLeft="10sp"
            android:text="Your Name"
            android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />

        <TextView
            android:id="@+id/textViewShowName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:layout_marginStart="50sp"
            android:layout_marginLeft="50sp"
            android:textAppearance="@style/Base.TextAppearance.AppCompat.Medium" />
    </TableRow>
</TableLayout>

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginBottom="156dp"
    android:gravity="center"
    android:orientation="horizontal"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
```

```
        app:layout_constraintStart_toStartOf="parent">

        <Button
            android:id="@+id/save"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Save" />

        <Button
            android:id="@+id/view"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="View" />

        <Button
            android:id="@+id/clear"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Clear" />
    </LinearLayout>

</android.support.constraint.ConstraintLayout>
```

MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    private val sharedPrefFile = "kotlinsharedpreference"
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val inputId = findViewById<EditText>(R.id.editId)
        val inputName = findViewById<EditText>(R.id.editName)
        val outputId = findViewById<TextView>(R.id.textViewShowId)
        val outputName = findViewById<TextView>(R.id.textViewShowName)

        val btnSave = findViewById<Button>(R.id.save)
        val btnView = findViewById<Button>(R.id.view)
        val btnClear = findViewById<Button>(R.id.clear)
        val sharedPreferences: SharedPreferences = this.getSharedPreferences(sharedPrefFile,
Context.MODE_PRIVATE)
        btnSave.setOnClickListener(View.OnClickListener {
            val id:Int = Integer.parseInt(inputId.text.toString())
            val name:String = inputName.text.toString()
            val editor: SharedPreferences.Editor =  sharedPreferences.edit()
            editor.putInt("id_key",id)
            editor.putString("name_key",name)
            editor.apply()
            editor.commit()
        })
        btnView.setOnClickListener {
            val sharedIdValue = sharedPreferences.getInt("id_key",0)
            val sharedNameValue = sharedPreferences.getString("name_key","defaultname")
            if(sharedIdValue.equals(0) && sharedNameValue.equals("defaultname")){
                outputName.setText("default name: ${sharedNameValue}").toString()
                outputId.setText("default id: ${sharedIdValue.toString()}")
            }else{
                outputName.setText(sharedNameValue).toString()
```

```
            outputId.setText(sharedIdValue.toString())
        }

    }
    btnClear.setOnClickListener(View.OnClickListener {
        val editor = sharedPreferences.edit()
        editor.clear()
        editor.apply()
        outputName.setText("").toString()
        outputId.setText("".toString())
    })
  }
}
```

❖ Internal Storage

→ **Android Internal Storage** is the device memory in which we store the files. The file stored in the internal storage is private in default, and only the same application accesses it. They cannot be accessed from outside the application.

→ To read and write the data from (into) the file, Android provides **openFileInput()** and **openFileOutput()** methods respectively.

→ When the users uninstall its application from the device, its internal storage file will also be removed.

➢ Writing file

→ To write the file in internal storage of device, **java.io** package offers **openFileOutput()** method which returns the instance of **FileOutputStream** class. To write the data into the file call the **FileOutputStream .write()** method.

```
val file:String = fileName.text.toString()
val data:String = fileData.text.toString()
val fileOutputStream:FileOutputStream
try {
    fileOutputStream = openFileOutput(file, Context.MODE_PRIVATE)
    fileOutputStream.write(data.toByteArray())
}catch (e: Exception){
    e.printStackTrace()
}
```

→ Methods of FileOutputStream class:

Method	Description
getChannel()	This method returns a write-only FileChannel that shares its position with this stream
getFD()	This method returns the underlying file descriptor
write(byte[] buffer, int byteOffset, int byteCount)	This method Writes count bytes from the byte array buffer starting at position offset to this stream
close()	This method is used to close file.

➢ Reading file

→ To read the file from the internal storage of device, **java.io** package offers **openFileInput()** method which returns the instance of **FileInputStream** class. To read the data from file call the **BufferedReader().readLine().**

```
var fileInputStream: FileInputStream? = null
fileInputStream = openFileInput(filename)
```

```
var inputStreamReader: InputStreamReader = InputStreamReader(fileInputStream)
val bufferedReader: BufferedReader = BufferedReader(inputStreamReader)
val stringBuilder: StringBuilder = StringBuilder()
var text: String? = null
while ({ text = bufferedReader.readLine(); text }() != null) {
   stringBuilder.append(text)
}
//Displaying data on EditText
fileData.setText(stringBuilder.toString()).toString()
```

→ **Methods of FileInputStream class**

Method	Description
available()	This method returns an estimated number of bytes that can be read or skipped without blocking for more input
getChannel()	This method returns a read-only FileChannel that shares its position with this stream
getFD()	This method returns the underlying file descriptor
read(byte[] buffer, int byteOffset, int byteCount)	This method reads at most length bytes from this stream and stores them in the byte array b starting at offset
close()	This method is used to close file.

❖ External Storage

→ Android External Storage is the memory space in which we perform read and write operation. Files in the external storage are stored in **/sdcard** or **/storage** folder etc. The files which are saved in the external storage is readable and can be modified by the user.

→ Before accessing the file in external storage in our application, we should check the availability of the external storage in our device.

→ It is necessary to add external storage read and write permission. For that we need to add permission in Android|Manifest.xml file.

-For writing:

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

-For reading:

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

➢ Writing file

→ The java.io package offers **openFileOutput()** method which returns the instance of **FileOutputStream** class to write the file in external storage of the device. To acquire a directory that's used by only our app by calling **getExternalFilesDir()**. To write the data into the file call the **FileOutputStream .write()** method.

```
var myExternalFile:File = File(getExternalFilesDir(filepath),fileName)
 try {
    val fileOutPutStream = FileOutputStream(myExternalFile)
    fileOutPutStream.write(fileContent.toByteArray())
    fileOutPutStream.close()
 } catch (e: IOException) {
    e.printStackTrace()
 }
```

→ Methods of FileOutputStream class:

Method	Description
getChannel()	This method returns a write-only FileChannel that shares its position with this stream

getFD()	This method returns the underlying file descriptor
write(byte[] buffer, int byteOffset, int byteCount)	This method Writes count bytes from the byte array buffer starting at position offset to this stream
close()	This method is used to close file.

➢ Reading file

→ The java.io package offers openFileInput() method which returns the instance of FileInputStream class and read the file from the external storage of the device. To read the data from file call the BufferedReader().readLine().

```
var myExternalFile:File = File(getExternalFilesDir(filepath), fileName)
val filename = fileName.text.toString()
myExternalFile = File(getExternalFilesDir(filepath),filename)
   var fileInputStream =FileInputStream(myExternalFile)
   var inputStreamReader: InputStreamReader = InputStreamReader(fileInputStream)
   val bufferedReader: BufferedReader = BufferedReader(inputStreamReader)
   val stringBuilder: StringBuilder = StringBuilder()
   var text: String? = null
   while ({ text = bufferedReader.readLine(); text }() != null) {
      stringBuilder.append(text)
   }
   fileInputStream.close()
```

→ **Methods of FileInputStream class**

Method	Description
available()	This method returns an estimated number of bytes that can be read or skipped without blocking for more input
getChannel()	This method returns a read-only FileChannel that shares its position with this stream
getFD()	This method returns the underlying file descriptor
read(byte[] buffer, int byteOffset, int byteCount)	This method reads at most length bytes from this stream and stores them in the byte array b starting at offset
close()	This method is used to close file.

→ **Methods to Store data in External Storage**

- **getExternalStoragePublicDirectory():** This is the present recommended method to keep files public and these files are not deleted even when the app is uninstalled from the system. For eg: Images clicked by the camera are still available even after we uninstall the camera.

- **getExternalFilesDir(String type):** This method is used to store private data that are specific to the app only. And data are removed as we uninstall the app.

- **getExternalStorageDirectory():** This method is not recommended. It is now absolute and it is used to access external storage in older versions, API Level less than 7.

→ The data files saved over external storage devices are publicly accessible on shared external storage using USB mass storage transfer. Data files stored over external storage using a **FileOutputStream** object and can be read using a **FileInputStream** object.(Detail available in Internal Storage)

❖ SQLite Database

→ SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.

→ SQLite supports all the relational database features. In order to access this database, we don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

→ It is embedded in android bydefault. So, there is no need to perform any database setup or administration task.

→ To use a SQLite database in an Android application, it is necessary to create a class that inherits from the SQLiteOpenHelper class, a standard Android class that arranges to open the database file. This class is available in "android.database.SQLite" package.

class DBHelper(context: Context, factory: SQLiteDatabase.CursorFactory?) :
SQLiteOpenHelper(context, DATABASE_NAME, factory, DATABASE_VERSION){}

➢ SQLiteOpenHelper:

→ It checks for the existence of the database file and, if it exists, it opens it; otherwise, it creates one

→ There are two constructors of SQLiteOpenHelper class:

Constructor	Detail
SQLiteOpenHelper(context: Context, name: String, factory: SQLiteDatabase.CursorFactory, version: Int)	creates an object for creating, opening and managing the database.
SQLiteOpenHelper(context: Context, name: String, factory: SQLiteDatabase.CursorFactory, version: Int, errorHandler: DatabaseErrorHandler)	creates an object for creating, opening and managing the database. It specifies the error handler.

→ **Methods:**

Method	Detail
onCreate()	Called when the database is created for the first time. This is where the creation of tables and the initial population of the tables should happen. **abstract fun onCreate(db: SQLiteDatabase!): Unit**
onUpgrade()	Called when the database needs to be upgraded. The implementation should use this method to drop tables, add tables, or do anything else it needs to upgrade to the new schema version. **abstract fun onUpgrade(db: SQLiteDatabase!, oldVersion: Int, newVersion: Int) : Unit**
onDowngrade()	Called when the database needs to be downgraded. This is strictly similar to onUpgrade(SQLiteDatabase, int, int) method, but is called whenever current version is newer than requested one. **onDowngrade(db: SQLiteDatabase!, oldVersion: Int, newVersion: I nt): Unit**
onOpen()	Called when the database has been opened. The implementation should check SQLiteDatabase#isReadOnly before updating the database. **onOpen(db: SQLiteDatabase!): Unit**
close()	Close any open database object. **close(): Unit**
onConfigure()	Called when the database connection is being configured, to enable features such as write-ahead logging or foreign key support. **onConfigure(db: SQLiteDatabase!): Unit**

getDatabaseName()	Return the name of the SQLite database being opened, as given to the constructor. **getDatabaseName(): String!**
getWritableDatabase()	Create and/or open a database that will be used for reading and writing. The first time this is called, the database will be opened and onCreate(SQLiteDatabase), onUpgrade(SQLiteDatabase, int, int) and/or onOpen(SQLiteDatabase) will be called. **getWritableDatabase(): SQLiteDatabase!**
getReadableDatabase()	Create and/or open a database. This will be the same object returned by getWritableDatabase() unless some problem, such as a full disk, requires the database to be opened read-only. In that case, a read-only database object will be returned. **getReadableDatabase(): SQLiteDatabase!**
setIdleConnectionTimeout()	Sets the maximum number of milliseconds that SQLite connection is allowed to be idle before it is closed and removed from the pool. This method should be called from the constructor of the subclass, before opening the database **setIdleConnectionTimeout(idleConnectionTimeoutMs: Long): Unit**
setWriteAheadLoggingEnabled()	Enables or disables the use of write-ahead logging for the database. Write-ahead logging cannot be used with read-only databases so the value of this flag is ignored if the database is opened read-only. **open Unit setWriteAheadLoggingEnabled(enabled: Boolean): Unit**

→ To create a table in SQL, we use the CREATE TABLE statement. Note that Android's API for SQLite assumes that our primary key will be a long integer (long in Java). The primary key column can have any name for now, but when we wrap the data in a ContentProvider this column is required to be named _id, so we'll start on the right foot by using that name now:

CREATE TABLE some_table_name (_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name TEXT);

→ Example:

```
class DBHelper(context: Context): SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {
   override fun onCreate(db: SQLiteDatabase) {
      val query = ("CREATE TABLE " + TABLE_NAME + " ("
            + ID_COL + " INTEGER PRIMARY KEY, " +
            NAME_COl + " TEXT," +
            EMAIL_COL + " TEXT" + ")")
      db.execSQL(query)
   }
   override fun onUpgrade(db: SQLiteDatabase, p1: Int, p2: Int) {
      // this method is to check if table already exists
      db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME)
      onCreate(db)
   }
companion object{
```

```
        private val DATABASE_NAME = "StudentsDB"
        private val DATABASE_VERSION = 1
        val TABLE_NAME = "student"
        val ID_COL = "id"
        val NAME_COl = "name"
        val EMAIL_COL = "email"
    }
}
```

➢ SQLiteDatabase:

→ SQLiteDatabase has methods to create, delete, execute SQL commands, and perform other common database management tasks.

→ **Methods:**

Method	Detail
execSQL(sql: String!): Unit	executes the sql query not select query. **Parameters** **sql:** the SQL statement to be executed. Multiple statements separated by semicolons are not supported.
insert(table: String!, nullColumnHack: String!,values: ContentValues!) :Long	inserts a record on the database. The table specifies the table name, nullColumnHack doesn't allow completely null values. If second argument is null, android will store null values if values are empty. The third argument specifies the values to be stored. **Parameters** **table :** Name of the Table **nullColumnHack:** optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If our provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where our values is empty. **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values.
update(table: String!,values: ContentValues!,whereClause: String!,whereArgs: Array<String!>!): Int	updates a row. **Parameters** **table:** Name of the Table **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values. **whereClause:** the optional WHERE clause to apply when updating. Passing null will update all rows. **whereArgs:** We may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.

delete(table: String!,whereClause: String!,whereArgs: Array<String!>!): Int	Used to delete table. **Parameters** **table:** Name of the Table **whereClause:** the optional WHERE clause to apply when updating. Passing null will delete all rows. **whereArgs:** We may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.
query(table: String!, columns: Array<String!>!, selection: String!, selectionArgs: Array<String!>!, groupBy: String!, having: String!, orderBy: String!) :Cursor!	returns a cursor over the resultset. **Parameters** **table:** Table Name **columns:** A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used. **selection:** A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table. **selectionArgs:** We may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings. **groupBy:** A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped. **having:** A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used. **orderBy:** How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.
query(table: String!, columns: Array<String!>!, selection: String!, selectionArgs: Array<String!>!, groupBy: String!, having: String!, orderBy: String!, limit: String!) :Cursor!	returns a cursor over the resultset. **Parameters** **limit:** Limits the number of rows returned by the query, formatted as LIMIT clause. Passing null denotes no LIMIT clause. Other parameters are describe above.
query(distinct: Boolean, table: String!, columns: Array<String!>!, selection: String!, selectionArgs: Array<String!>!, groupBy: String!, having: String!, orderBy: String!, limit: String!) :Cursor!	returns a cursor over the resultset. **Parameters** **distinct:** true if we want each row to be unique, false otherwise. Other parameters are describe above.
rawQuery (sql: String!, selectionArgs: Array<String!>!) :Cursor!	Runs the provided SQL and returns a Cursor over the result set. **Parameters** **sql:** the SQL statement to be executed. Multiple statements separated by semicolons are not supported. **selectionArgs:** We may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.

releaseMemory() :Int	Attempts to release memory that SQLite holds but does not require to operate properly. Typically this memory will come from the page cache.
replace(table: String!, nullColumnHack: String!, initialValues: ContentValues!): Long	Convenience method for replacing a row in the database. Inserts a new row if a row does not already exist. **Parameters** **table :** Name of the Table **nullColumnHack:** optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If our provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where our values is empty. **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values.
isOpen (): Boolean	Returns true if the database is currently open.
isReadOnly (): Boolean	Returns true if the database is opened as read only.
isWriteAheadLoggingEnabled (): Boolean	Returns true if write-ahead logging has been enabled for this database.
deleteDatabase (File file): Boolean	Deletes a database including its journal file and other auxiliary files that may have been created by the database engine. **Parameters** file: The database file path. This value cannot be null.
getMaximumSize (): Long	Returns the maximum size the database may grow to.
getPageSize (): Long	Returns the current database page size, in bytes.
getPath(): String!	Gets the path to the database file.
getVersion(): Int	Gets the database version.

➢ Cursor interface:

→ This interface provides random read-write access to the result set returned by a database query.

→ Cursor implementations are not required to be synchronized so code using a Cursor from multiple threads should perform its own synchronization when using the Cursor.

→ **Methods:**

Method	Detail
abstract fun close(): Unit	Closes the Cursor, releasing all of its resources and making it completely invalid. Unlike deactivate() a call to requery() will not make the Cursor valid again.
abstract fun getColumnCount(): Int	Return total number of columns

abstract fun getColumnIndex(columnName: String!): Int	Returns the zero-based index for the given column name, or -1 if the column doesn't exist. If we expect the column to exist use getColumnIndexOrThrow(java.lang.String) instead, which will make the error more clear. **Parameters** **columnName:** the name of the target column.
abstract fun getColumnName (columnIndex: Int): String!	Returns the column name at the given zero-based column index. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getColumnNames(): Array<String!>!	Returns a string array holding the names of all of the columns in the result set in the order in which they were listed in the result.
abstract fun getCount(): Int	Returns the numbers of rows in the cursor.
abstract fun getDouble (columnIndex: Int): Double	Returns the value of the requested column as a double. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract float getFloat (int columnIndex)	Returns the value of the requested column as a float. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getInt (columnIndex: Int): Int	Returns the value of the requested column as an int. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getLong (columnIndex: Int): Long	Returns the value of the requested column as a long. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getShort (columnIndex: Int): Short	Returns the value of the requested column as a short. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getString (columnIndex): String!	Returns the value of the requested column as a String. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getType (columnIndex: Int): Int	Returns data type of the given column's value. The preferred type of the column is returned but the data may be converted to other types as documented in the get-type methods such as getInt(int), getFloat(int) etc. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
abstract fun getExtras (): Bundle!	Returns a bundle of extra values. This is an optional way for cursors to provide out-of-band metadata to their users. One use of this is for reporting on the progress of network requests that are required to fetch data for the cursor.
abstract fun getPosition(): Int	Returns the current position of the cursor in the row set. The value is zero-based. When the row set is first returned the cursor will be at positon -1, which is before the first row. After the last row is returned another call to next() will leave the cursor past the last entry, at a position of count().

abstract fun move (offset: Int): Boolean	Move the cursor by a relative amount, forward or backward, from the current position. Positive offsets move forwards, negative offsets move backwards. If the final position is outside of the bounds of the result set then the resultant position will be pinned to -1 or count() depending on whether the value is off the front or end of the set, respectively. **Parameter** **offset:** the offset to be applied from the current position.
abstract fun moveToFirst (): Boolean	Move the cursor to the first row.
abstract fun moveToLast (): Boolean	Move the cursor to the last row.
abstract fun moveToNext (): Boolean	Move the cursor to the next row.
abstract fun moveToPrevious (): Boolean	Move the cursor to the previous row.
abstract fun moveToPosition (int position): Boolean	Move the cursor to an absolute position. The valid range of values is -1 <= position <= count. This method will return true if the request destination was reachable, otherwise, it returns false. **Parameter** **position:** The zero-based position to move to. Value is -1 or greater
abstract fun isAfterLast (): Boolean	Returns whether the cursor is pointing to the position after the last row.
abstract fun isBeforeFirst (): Boolean	Returns whether the cursor is pointing to the position before the first row.
abstract fun isClosed (): Boolean	return true if the cursor is closed.
abstract fun isFirst (): Boolean	Returns whether the cursor is pointing to the first row.
abstract fun isLast (): Boolean	Returns whether the cursor is pointing to the last row.
abstract fun isNull(columnIndex: Int): Boolean	Returns true if the value in the indicated column is null.

➢ CursorAdapter:

→ Android provides adapter classes specifically to display data from an SQLite database query. There is SimpleCursorAdapter class, which is more simpler and you cannot use your own custom xml layout and you don't have the control of the layout. In order to use custom xml layout, Android provides CursorAdapter.

→ A CursorAdapter makes it easy to use when the resource of a listview is coming from database and you can have more control over the binding of data values to layout controls. In the newView() method, you simply inflate the view and return it. In the bindView() method, you set the elements of your view.

→ Constructor:

```
CursorAdapter(context: Context!, c: Cursor!)
CursorAdapter(context: Context!, c: Cursor!, autoRequery: Boolean)
CursorAdapter(context: Context!, c: Cursor!, flags: Int)
```

→ Methods:

```
abstract fun bindView(view: View!, context: Context!, cursor: Cursor!): Unit
open fun changeCursor(cursor: Cursor!): Unit
open fun getCount(): Int
open fun getCursor(): Cursor!
open fun getItem(position: Int): Any!
open fun getItemId(position: Int): Long
open fun getView(position: Int, convertView: View!, parent: ViewGroup!): View!
abstract fun newView(context: Context!, cursor: Cursor!, parent: ViewGroup!): View!
open fun swapCursor(newCursor: Cursor!): Cursor!
```

➢ Schema Object:

→ This object represents a set of constraints that can be checked/ enforced against an XML document.
→ A Schema object is thread safe and applications are encouraged to share it across many parsers in many threads.
→ A Schema object is immutable in the sense that it shouldn't change the set of constraints once it is created. In other words, if an application validates the same document twice against the same Schema, it must always produce the same result.
→ A Schema object is usually created from SchemaFactory.
→ Constructor:

protected Schema()

→ Methods:

abstract fun newValidator(): Validator!

A validator enforces/checks the set of constraints this object represents.

abstract fun newValidatorHandler(): ValidatorHandler!

Creates a new ValidatorHandler for this Schema.

➢ Simple Adapter

→ In Android, whenever we want to bind some data which we get from any data source (e.g. ArrayList, HashMap, SQLite, etc.) with a UI component(e.g. ListView, GridView, etc.) then Adapter comes into the picture. Basically Adapter acts as a bridge between the UI component and data sources. Here Simple Adapter is one type of Adapter. It is basically an easy adapter to map static data to views defined in our XML file(UI component) and is used for customization of List or Grid items.
→ Constructor:

SimpleAdapter(context: Context!, data: MutableList<out MutableMap<String!, *>!>!, resource: Int, from: Array<String!>!, to: IntArray!)

→ Methods:

open fun getCount(): Int
open fun getItem(position: Int): Any!
open fun getItemId(position: Int): Long
open fun getView(position: Int, convertView: View!, parent: ViewGroup!): View!
open fun getViewBinder(): SimpleAdapter.ViewBinder!
open fun setViewBinder(viewBinder: SimpleAdapter.ViewBinder!): Unit
open fun setViewImage(v: ImageView!, value: Int): Unit
open fun setViewText(v: TextView!, text: String!): Unit

➢ Conrtact Class

→ A contract class is a public final class that contains constant definitions for the URIs, column names, MIME types, and other meta-data about the ContentProvider. It can also contain static helper methods to manipulate the URIs. In simple terms, Contract class is used by the developers to define a schema and have a convention where to find the database constants.