Storage Options

Introduction to Storage options

- Android provides several options for you to save persistent application data.
- The solution you choose depends on your specific needs.
- It depends on : whether the data should be private to your application or accessible to other applications.
- It also depends on how much space your data requires.

Storage options in Android

- **SQLite Databases:** Store structured data in a private database.
- **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.
- **Network Connection:** Store data on the web with your own network server.

SQLite Database

- Android provides full support for SQLite databases.
- Any databases you create will be accessible by name to any class in the application, but not outside the application.
- The recommended method to create a new SQLite database is to create a subclass of **SQLiteOpenHelper**

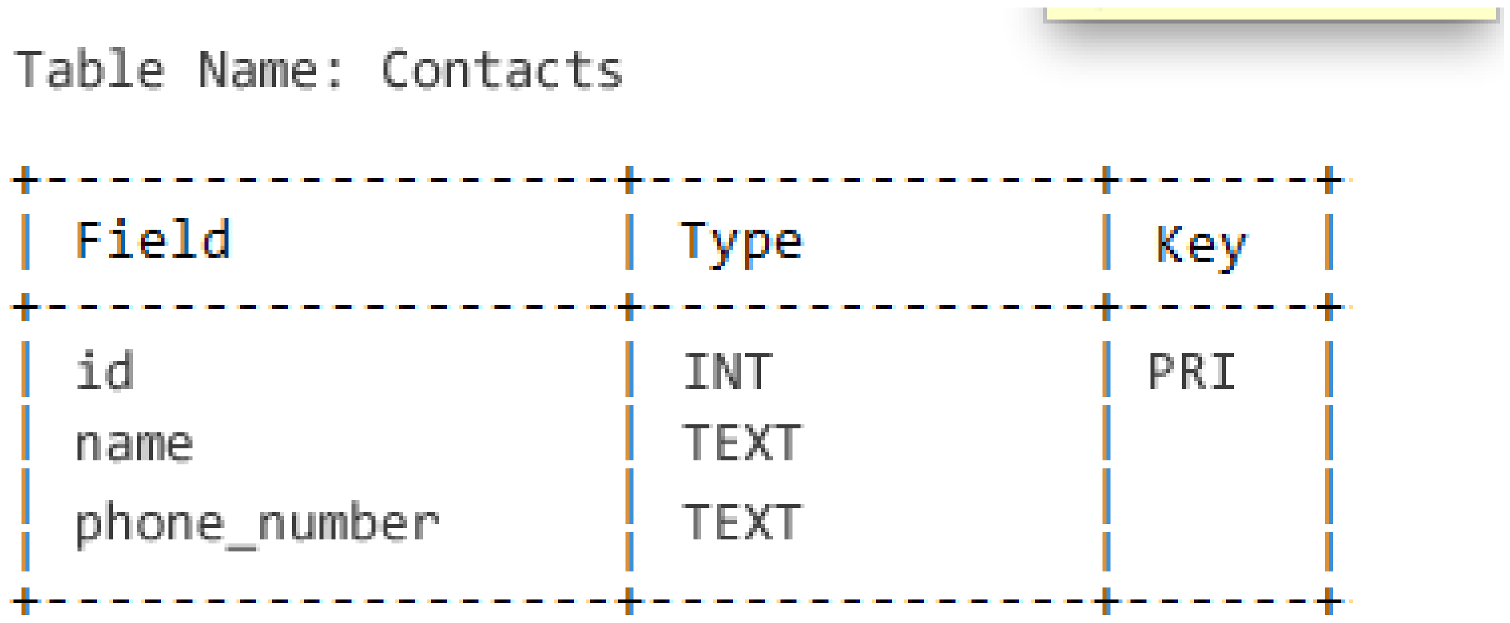
SQLite Database

- override the **onCreate() method:** It executes a SQLite command to create tables in the database.
- call **getWritableDatabase() and getReadableDatabase() to read and write in database.**
- These both return a **SQLiteDatabase object that represents the database and provides methods for SQLite operations.**

SQLite Database query

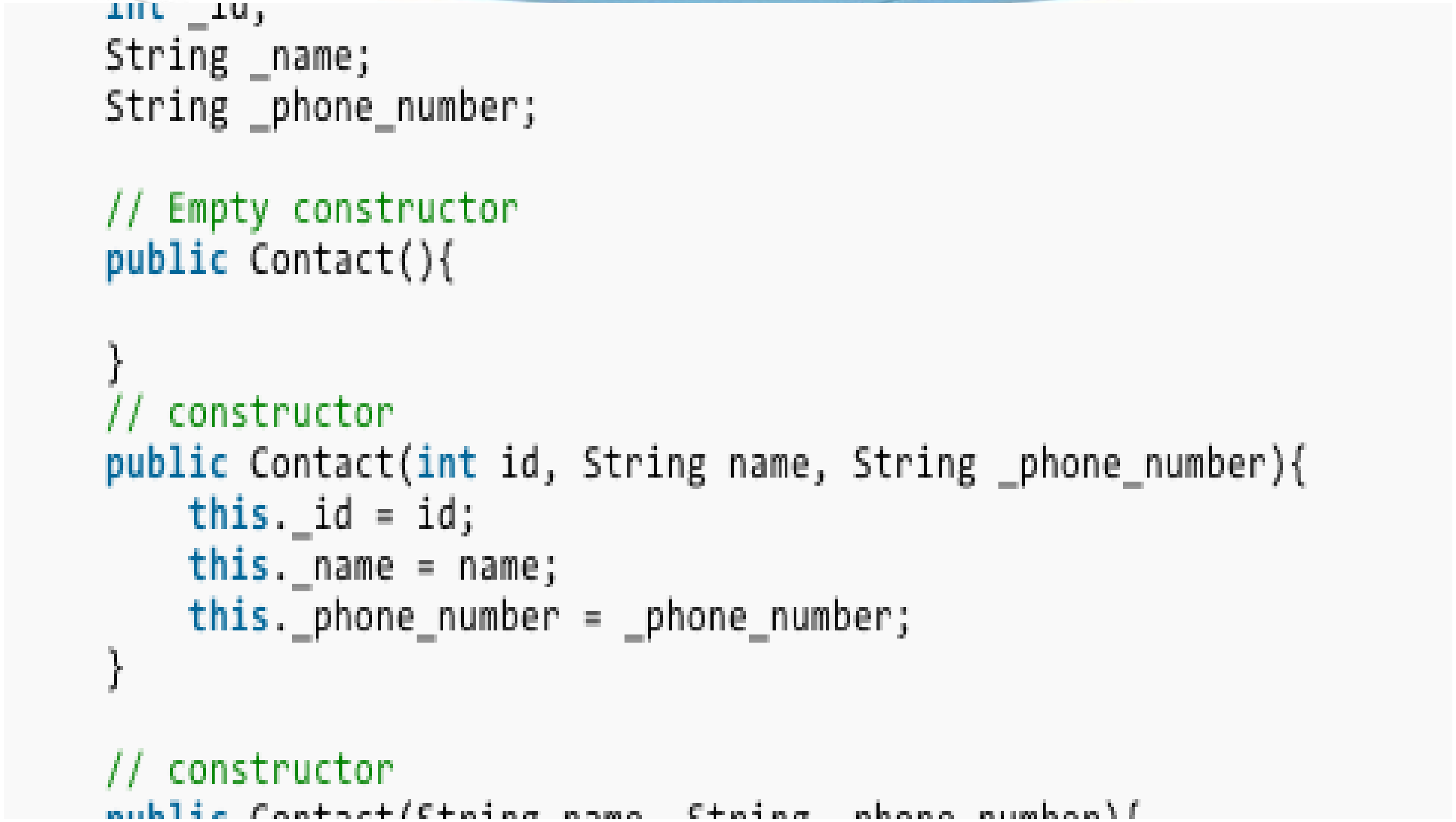
- Execute SQLite queries using the SQLiteDatabase **query() methods.**
- Accepts various query parameters, such as the table to query, the projection, selection, columns, grouping, and others.
- Every SQLite query will return a **Cursor that points to all the rows found by the query.**
- The **Cursor is always the mechanism with which you can navigate results from a database query and read rows and columns.**

Example



```
Table Name: Contacts

+--------------------+------------------+--------+
| Field              | Type             | Key    |
+--------------------+------------------+--------+
| id                 | INT              | PRI    |
| name               | TEXT             |        |
| phone_number       | TEXT             |        |
+--------------------+------------------+--------+
```

Step:1Writing Contact Class



```
int _id;
String _name;
String _phone_number;

// Empty constructor
public Contact(){

}
// constructor
public Contact(int id, String name, String _phone_number){
    this._id = id;
    this._name = name;
    this._phone_number = _phone_number;
}

// constructor
public Contact(String name, String _phone_number){
```

Step 2: Writing SQLite Database Handler Class

- **public class** DatabaseHandler **extends** SQLiteOpenHelper {

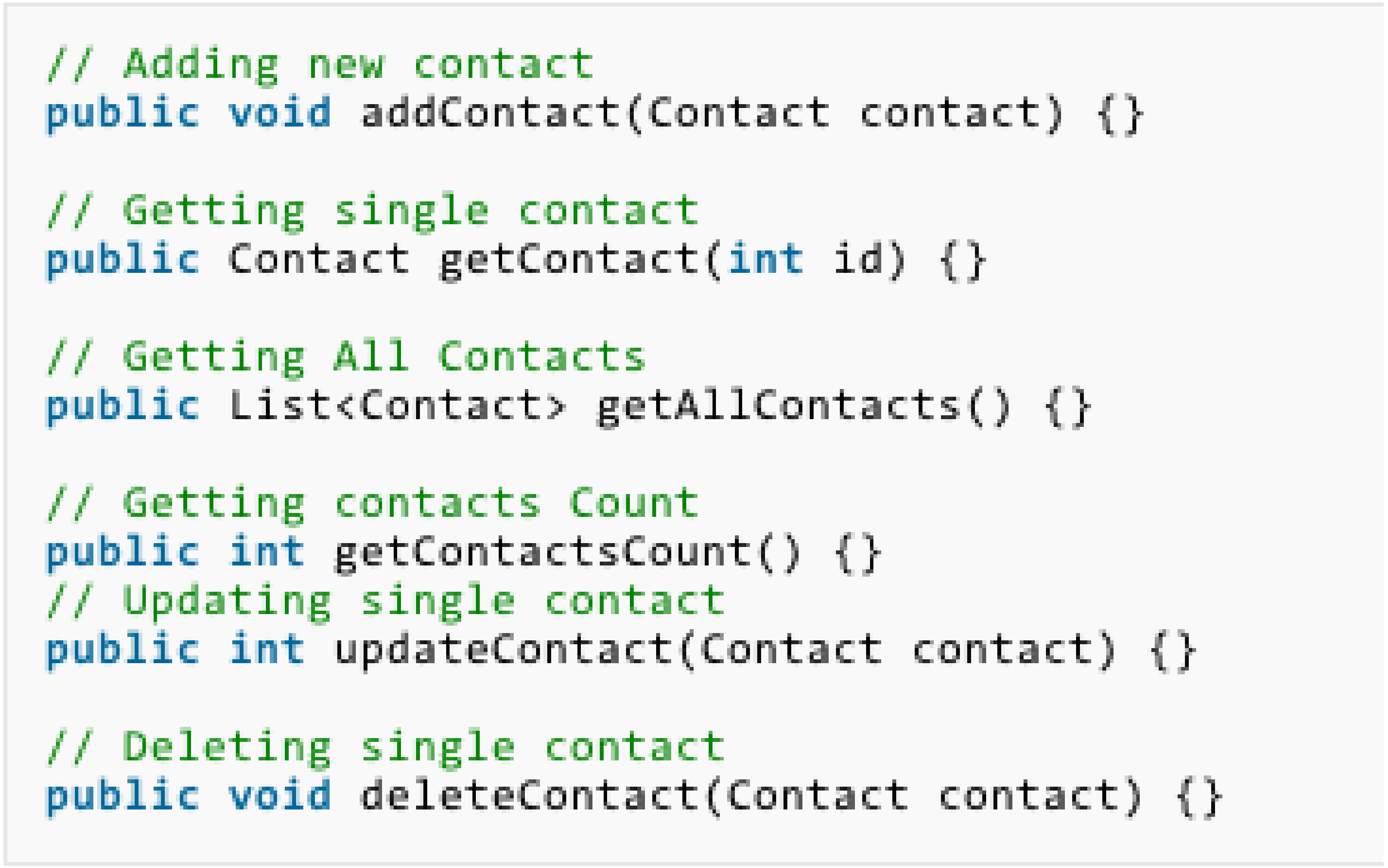
- Next call **onCreate()** and **onUpgrage():**

onCreate() –This is called when database is created.

onUpgrade() – This method is called when database is upgraded like modifying the table structure, adding constraints to database etc.,

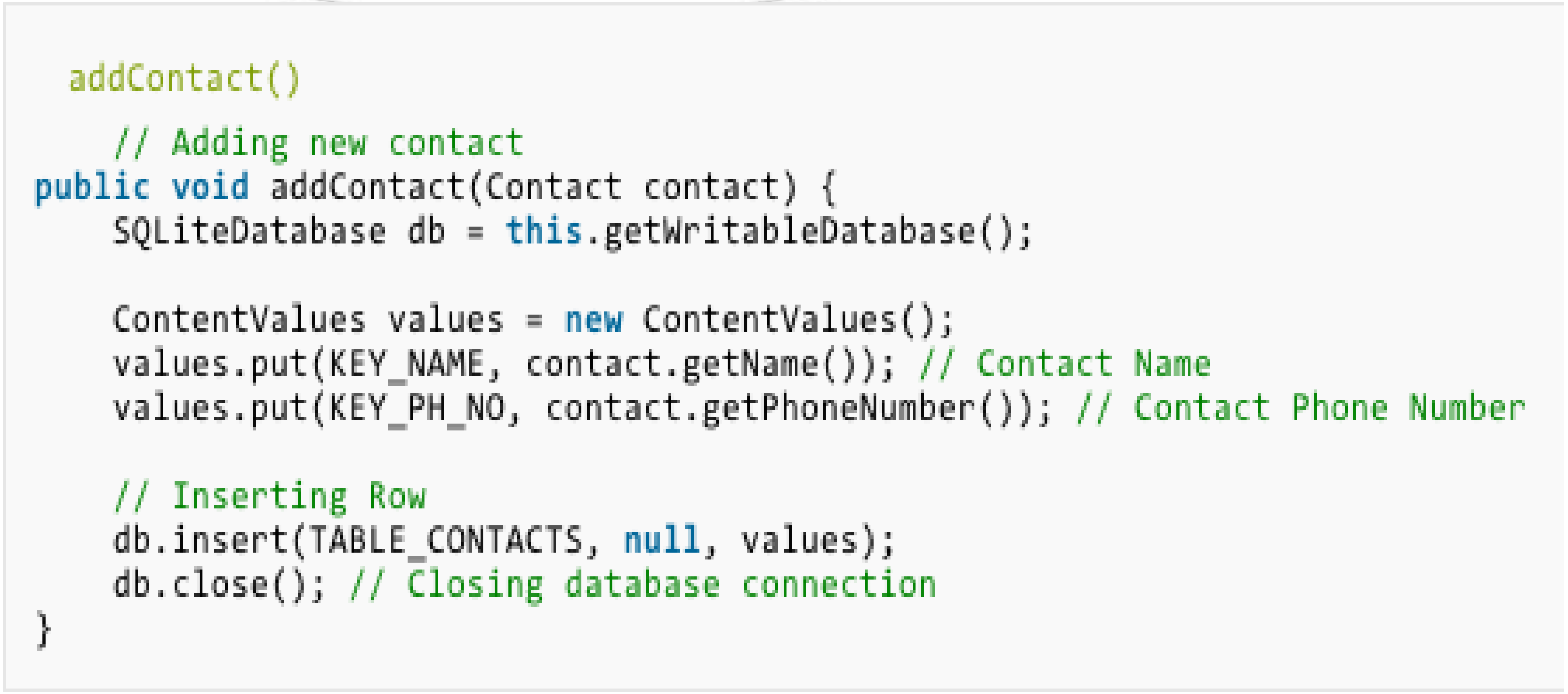
Step 3: CRUD Operation

(Create, Read, Update, Delete)



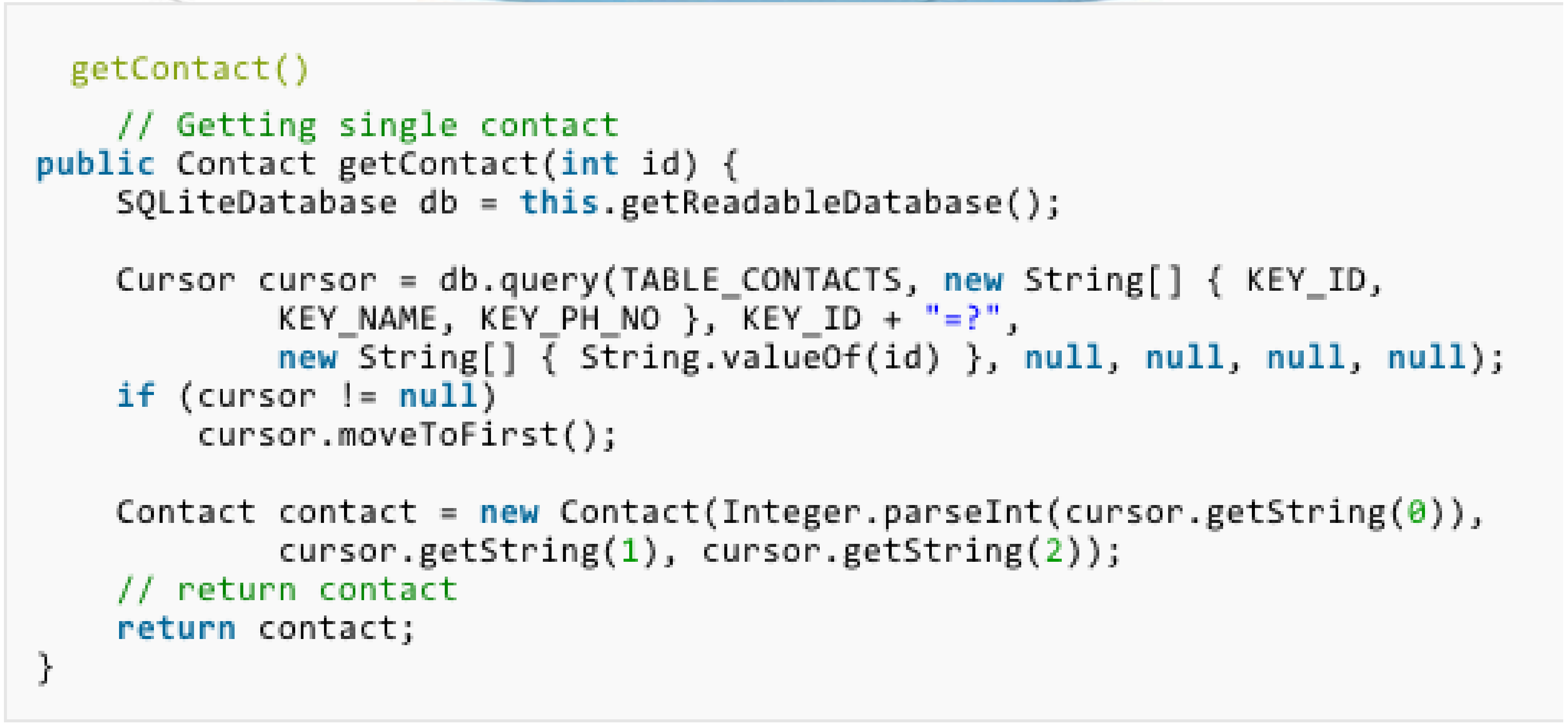
```
// Adding new contact
public void addContact(Contact contact) {}

// Getting single contact
public Contact getContact(int id) {}

// Getting All Contacts
public List<Contact> getAllContacts() {}

// Getting contacts Count
public int getContactsCount() {}
// Updating single contact
public int updateContact(Contact contact) {}

// Deleting single contact
public void deleteContact(Contact contact) {}
```

⇒Inserting new Record



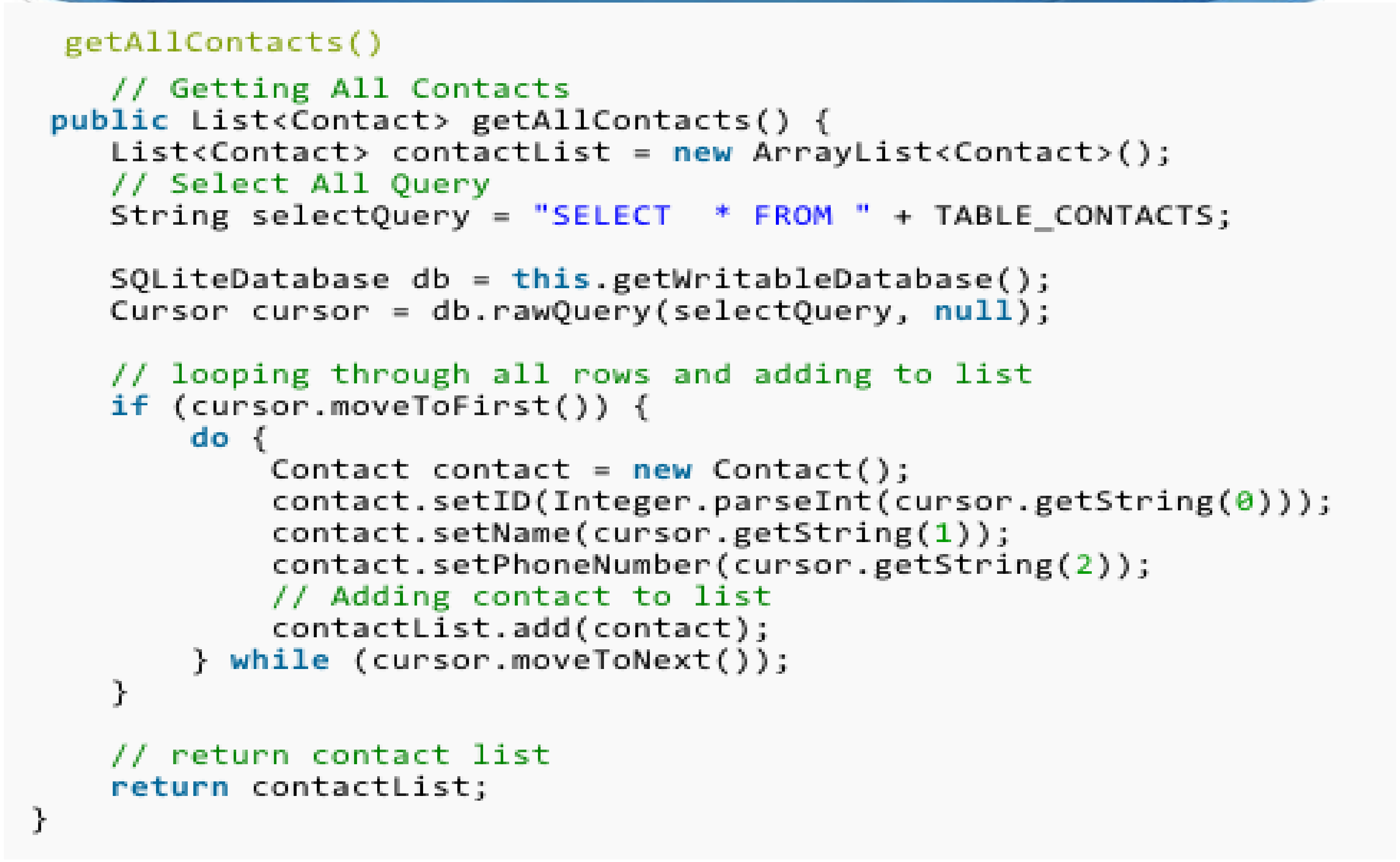
```
  addContact()

    // Adding new contact
public void addContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_NAME, contact.getName()); // Contact Name
    values.put(KEY_PH_NO, contact.getPhoneNumber()); // Contact Phone Number

    // Inserting Row
    db.insert(TABLE_CONTACTS, null, values);
    db.close(); // Closing database connection
}
```

⇒Reading Row(s)



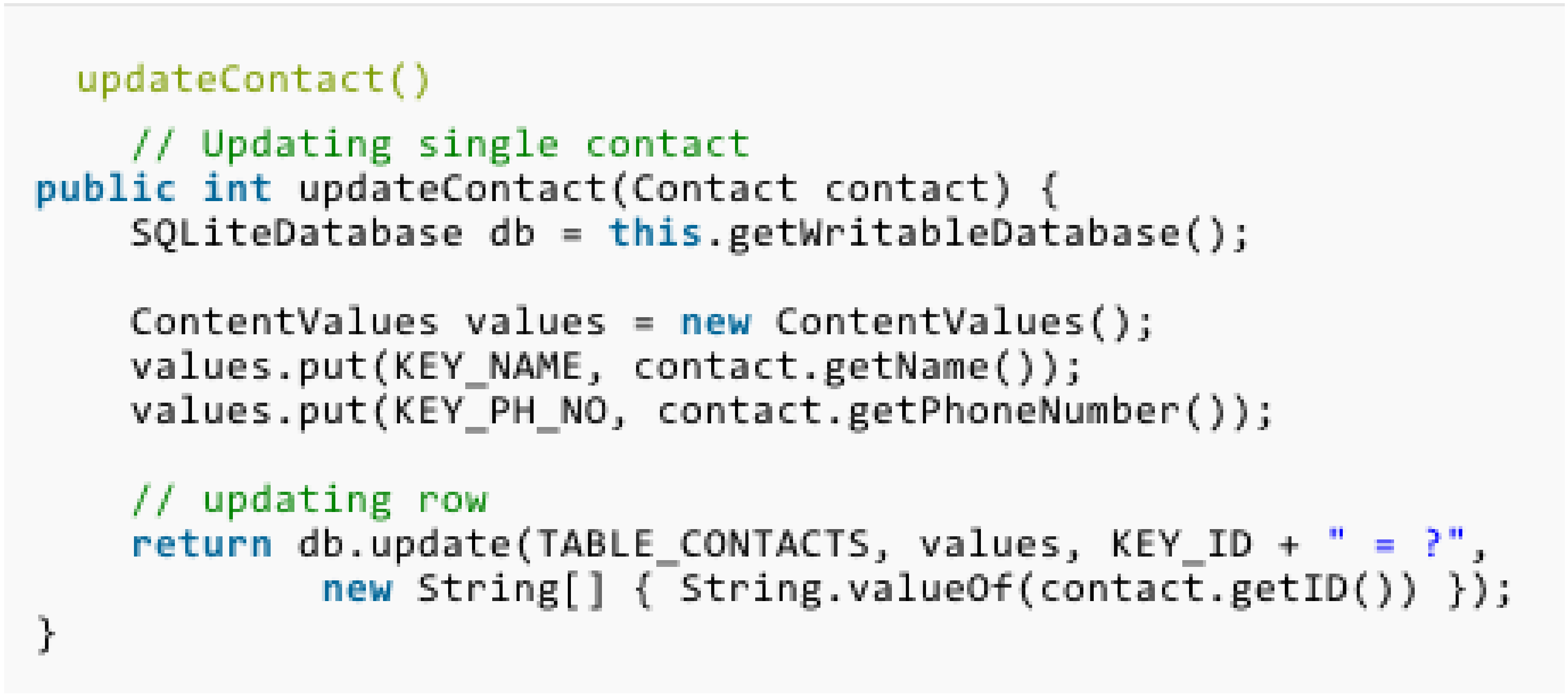
```
getContact()

    // Getting single contact
public Contact getContact(int id) {
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.query(TABLE_CONTACTS, new String[] { KEY_ID,
            KEY_NAME, KEY_PH_NO }, KEY_ID + "=?",
            new String[] { String.valueOf(id) }, null, null, null, null);
    if (cursor != null)
        cursor.moveToFirst();

    Contact contact = new Contact(Integer.parseInt(cursor.getString(0)),
            cursor.getString(1), cursor.getString(2));
    // return contact
    return contact;
}
```

=> GetAllContacts



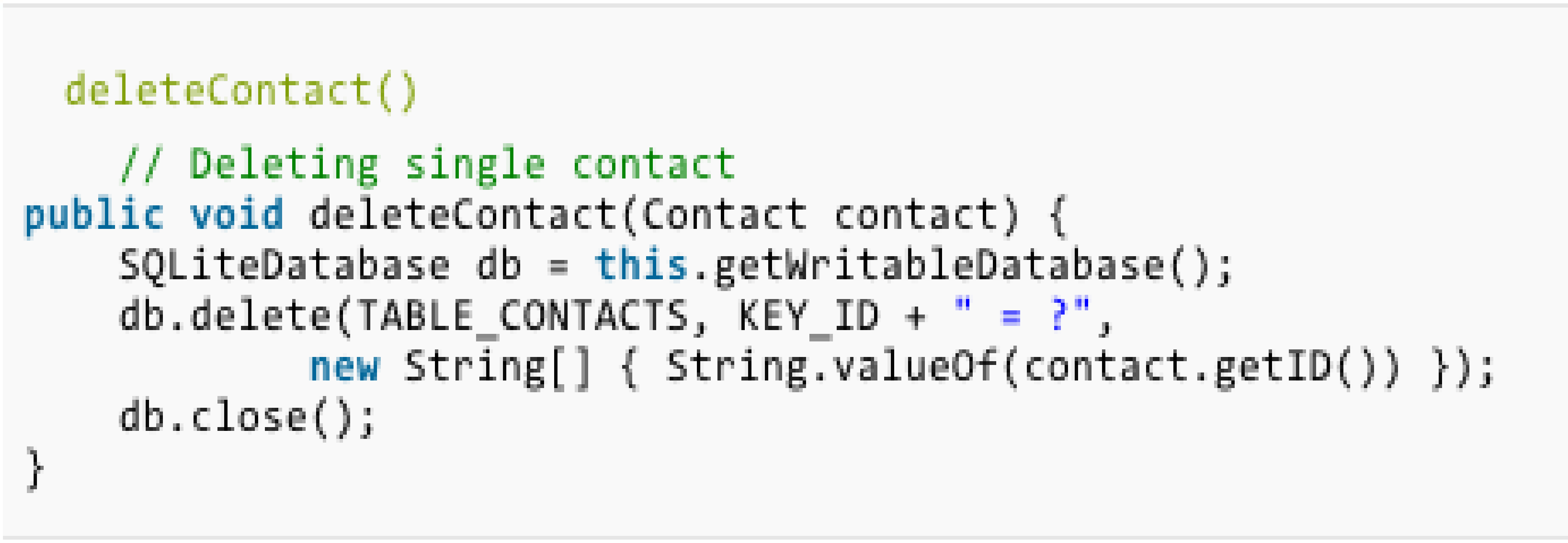
```
 getAllContacts()
    // Getting All Contacts
 public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<Contact>();
    // Select All Query
    String selectQuery = "SELECT  * FROM " + TABLE_CONTACTS;

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    // looping through all rows and adding to list
    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact();
            contact.setID(Integer.parseInt(cursor.getString(0)));
            contact.setName(cursor.getString(1));
            contact.setPhoneNumber(cursor.getString(2));
            // Adding contact to list
            contactList.add(contact);
        } while (cursor.moveToNext());
    }

    // return contact list
    return contactList;
}
```

⇒Updating Record



```
  updateContact()

    // Updating single contact
public int updateContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_NAME, contact.getName());
    values.put(KEY_PH_NO, contact.getPhoneNumber());

    // updating row
    return db.update(TABLE_CONTACTS, values, KEY_ID + " = ?",
            new String[] { String.valueOf(contact.getID()) });
}
```

⇒Deleting Record



```
deleteContact()

    // Deleting single contact
public void deleteContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_CONTACTS, KEY_ID + " = ?",
            new String[] { String.valueOf(contact.getID()) });
    db.close();
}
```

Shared Preferences

- The SharedPreferences class provides a general framework that allows you to save and retrieve persistent **key-value** pairs of primitive data types.
- Use SharedPreferences to save any primitive data: **booleans, floats, ints, longs, and strings.**
- This data will persist across user sessions (even if your application is killed).

Using Shared Preferences

- Following two methods can be used for getting Shared Preferences:
- **getPreferences()** - Use this if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, you don't supply a name.
- **getSharedPreferences()** – Use this if you need multiple preferences files identified by name, which you specify with the first parameter.

Steps to write Values in Shared Preferences

- To write values:
  - Call **edit()** to get a SharedPreferences Editor.
  - Add values with methods such as **putBoolean() and putString().**
  - Commit the new values with **commit().**

```
SharedPreferences settings = getSharedPreferences(PREFS_NAME,
    Context.MODE_PRIVATE);

Editor editor = settings.edit();

editor.putBoolean("silentMode", mSilentMode);

editor.commit();
```

Steps to Read Values From Shared Preferences

- To read values, use SharedPreferences methods such as **getBoolean() and getString().**

```
SharedPreferences settings = getSharedPreferences(PREFS_NAME,
     Context.MODE_PRIVATE);
boolean silent = settings.getBoolean("silentMode", false);
if (silent)
{
     Log.v(“Message”,”Phone is in Silent”);
 }
```

Internal Storage

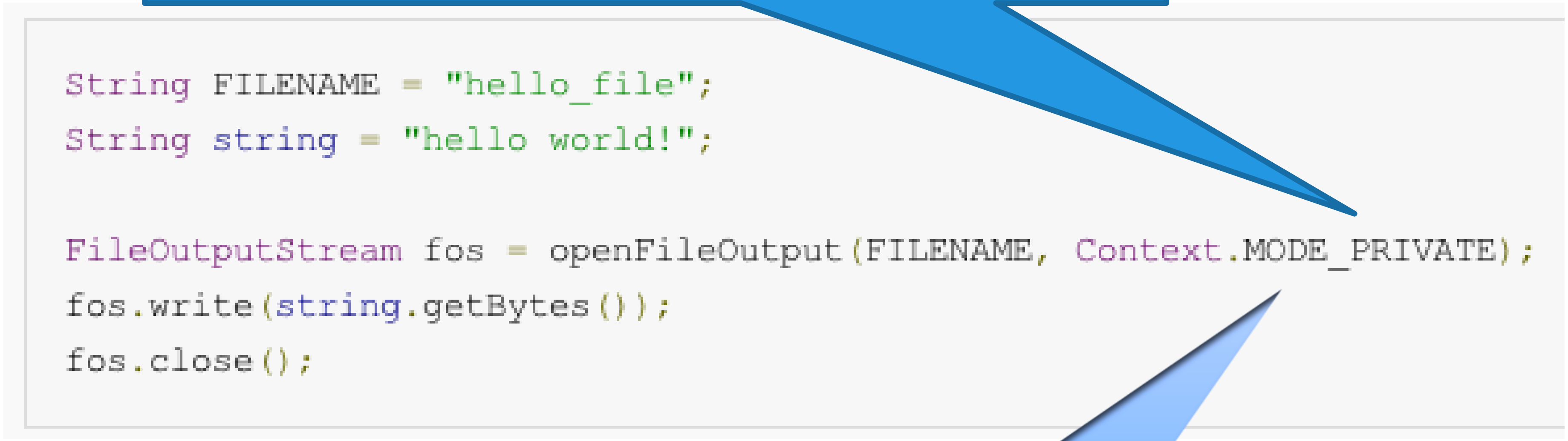
- Internal Storage in Android is similar to Files Handling in Other platforms.
- It allows to save files directly on the device's internal storage.
- By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user).
- When the user uninstalls your application, these files are removed.

Steps to create files

- To create and write a private file to the internal storage:

1. Call **openFileOutput()** with the name of the file and the operating mode. **This returns a FileOutputStream.**
2. Write to the file with **write().**
3. Close the stream with **close().**

MODE_APPEND, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE.



```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

MODE_PRIVATE will create the file (or replace a file of the same name) and make it private to your application.

To read from file

- **To read a file from internal storage:**
- Call **openFileInput()** and pass it the name of the file to read. This returns a **FileInputStream.**
- Read bytes from the file with **read().**
- Then close the stream with **close().**
- If you want to save a static file in your application at compile time, save the file in your project **res/raw/** directory. You can open it with **openRawResource()**, passing the **R.raw.<*filename*>** resource ID. This method returns an **InputStream** that you can use to read the file (but you cannot write to the original file).

External Storage

- If you want to save files that are not specific to your application and that should ***not*** be deleted when your application is uninstalled, save them to one of the public directories on the external storage.
- These directories lay at the root of the external storage, such as **Music/**, **Pictures/**, **Ringtones/**, and others.
- **getExternalStoragePublicDirectory()** method can be used to get access to all the external folders.
- Add **WRITE_EXTERNAL_STORAGE and READ_EXTERNAL_STORAGE** permission to the manifest file for writing and reading operations.

Use of External Storage

- External storage is useful for the educational or training based apps to store videos.
- To store large size media for games.
- Magazine apps to store the content of magazine pages.
- To store the large output files created by user.

Default Folders

- **Music/** - Media scanner classifies all media found here as user music.
- **Podcasts/** - Media scanner classifies all media found here as a podcast.
- **Ringtones/** - Media scanner classifies all media found here as a ringtone.
- **Alarms/** - Media scanner classifies all media found here as an alarm sound.
- **Notifications/** - Media scanner classifies all media found here as a notification sound.
- **Pictures/** - All photos (excluding those taken with the camera).
- **Movies/** - All movies (excluding those taken with the camcorder).
- **Download/** - Miscellaneous downloads.

Network Connection

- Use the network to store and retrieve data on your own web-based services.
- To do network operations, use classes in the following packages:

  android.net.*

  java.net.*

- To begin with URL parsing we need the API URL from where we need to parse the data.