Unit – 3 Data Access

Data Persistence

- Persisting data is an important issue in application development, as users typically expect to reuse data in the future.
- For Android, there are primarily three basic ways of persisting data:

1. A lightweight mechanism known as *shared preferences to save small chunks of data*
2. Traditional file systems
3. A relational database management system through the support of SQLite databases

Data Persistence

- Android provides the “**SharedPreferences**” object to help you save simple application data.
- For example, your application may have an option that enables users to specify the font size of the text displayed in your application.
- In this case, your application needs to remember the size set by the user so that the next time he or she uses the application again, it can set the size appropriately.

Data Persistence

- In order to do so, you have several options.

1. You can save the data to a file, but
   - I. you have to perform some file management routines, such as writing the data to the file, indicating how many characters to read from it, and so on. Also,
   - II. if you have several pieces of information to save, such as text size, font name, preferred background colour, and so on, then the task of writing to a file becomes more onerous.

Data Persistence

2. An alternative to writing to a text file is to use a database, but saving simple data to a database is overkill, both from a developer's point of view and in terms of the application's run-time performance.

- Using the ***SharedPreferences*** object, however, you save the data you want through the use of name/value pairs
  - specify a name for the data you want to save, and then both it and its value will be saved automatically to an XML file for you.

Shared Preferences

- To use the SharedPreferences object, you use the ***getSharedPreferences()*** method, passing it the name of the shared preferences file in which all the data will be saved, as well as the mode in which it should be opened:

```
private SharedPreferences prefs;
//---get the SharedPreferences object---
prefs = getSharedPreferences(prefName, MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
```

- The MODE_PRIVATE constant indicates that the shared preference file can only be opened by the application that created it.
- The Editor class allows you to save key/value pairs to the preferences file.

Shared Preferences

- When you are done saving the values, call the ***commit()*** method to save the changes:

```
//---save the values in the EditText view to preferences---
editor.putFloat(FONT_SIZE_KEY, editText.getTextSize());
editor.putString(TEXT_VALUE_KEY, editText.getText().toString());
//---saves the values---
editor.commit();
```

- The shared preferences file is saved as an XML file in the */data/data/<package_name>/shared_prefs* folder

Shared Preferences

- When the activity is loaded, you first obtain the SharedPreferences object and then retrieve all the values saved earlier:

```
//---load the SharedPreferences object---
SharedPreferences prefs =
    getSharedPreferences(prefName,
    MODE_PRIVATE);
```

Shared Preferences

- The information saved inside the SharedPreferences object is visible to all the activities within the same application.
- However, if you don't need to share the data between activities, you can use the ***getPreferences()*** method, like this:

```
//---get the SharedPreferences object---
prefs = getPreferences(MODE_PRIVATE);
```

- The ***getPreferences()*** method does not require a name, and the data saved is restricted to the activity that created it.
- In this case, the filename used for the preferences file will be named after the activity that created it

Persisting Data To Files

- The SharedPreferences object enables you to store data that is best stored as name/value pairs
  - for example, user ID, birth date, gender, driving license number, and so on.
- However, sometimes you might prefer to use the traditional file system to store your data.
- For example, you might want to store the text of poems you want to display in your applications.
- In Android, you can use the classes in the “***java.io***” package to do so.

Persisting Data To Files

- The first way to save files in your Android application is to write to the device's internal storage.
- Sometimes, it would be useful to save data to external storage (such as an SD card) because of its larger capacity, as well as the capability to share the files easily with other users (by removing the SD card and passing it to somebody else).

Persisting Data To Files

- To save text into a file, you use the **FileOutputStream** class.
- The **openFileOutput()** method opens a named file for writing, with the mode specified.
- The MODE_WORLD_READABLE constant to indicate that the file is readable by all other applications:

  FileOutputStream fOut = openFileOutput("textfile.txt", MODE_WORLD_READABLE);

Persisting Data To Files

- Apart from the MODE_WORLD_READABLE constant, you can select from the following:
  1. MODE_PRIVATE (file can only be accessed by the application that created it),
  2. MODE_APPEND (for appending to an existing file), and
  3. MODE_WORLD_WRITEABLE (all other applications have write access to the file).

Persisting Data To Files

- Then, **write()** method is used to write the string to the file.
- To ensure that all the bytes are written to the file, use the **flush()** method.
- Finally, use the **close()** method to close the file:

    osw.write(str);

    osw.flush();

    osw.close();

Persisting Data To Files

- To read the content of a file, you use the **FileInputStream** class, together with the **InputStreamReader** class:

```
FileInputStream fIn =
          openFileInput(“textfile.txt”);
InputStreamReader isr = new
          InputStreamReader(fIn);
```

Persisting Data To Files

- **StringBuffer** and **BufferedReader** objects are used to store data temporarily before sending it to the view for display.

```
BufferedReader bufferedReader = new BufferedReader(isr);
StringBuffer stringBuffer = new StringBuffer();
```

- An empty string is created in which the values from buffer reader are stored.

```
String lines;
while((lines = bufferedReader.readLine()) != null)
{
    stringBuffer.append(lines + "\n");
}
txtshow.setText(stringBuffer.toString());
```

Storing Data in Database

- For saving relational data, using a database is much more efficient.
- For example, if you want to store the test results of all the students in a school, it is much more efficient to use a database to represent them because you can use database querying to retrieve the results of specific students.
- Moreover, using databases enables you to enforce data integrity by specifying the relationships between different sets of data.
- Android uses the SQLite database system.
- The database that you create for an application is only accessible to itself; other applications will not be able to access it.
- SQLite database that you create programmatically in an application is always stored in the /data/data/<package_name>/ databases folder.

Android Databases

- Android provides structured data persistence through a combination of **SQLite databases** and **Content Providers**.
- SQLite databases can be used to store application data using a managed, structured approach.
- Android offers a full SQLite relational database library.
- Every application can create its own database over which it has complete control.

Content Providers

- Content Provider offer a generic, well-defined interface for using and sharing data, and provides a consistent abstraction from the underlying data source.
- They enable you to decouple your application layers from the underlying data layers, making your application data source agnostic by abstracting the underlying data source.
- Content Providers can be shared between applications, queried for results, have their existing records updated or deleted, and have new records added.

SQLite

- SQLite is a well-regarded relational database management system.
- SQLite is:
  - Open Source
  - Standard-Compliant
  - Lightweight
  - Single-tier
- It has been implemented as a compact C library that is included as part of the Android software stack.

SQLite

- By being implemented as a library, rather than running as a separate ongoing process, each SQLite database is an integrated part of the application that created it.
- This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.
- SQLite is extremely reliable and is the database system of choice for many consumer electronic devices.

SQLite

- Lightweight and powerful, SQLite differs from many conventional database engines by loosely typing each column, meaning that column values are not required to conform to a single type; instead, each value is typed individually in the each row.
- As a result, type checking isn’t necessary when assigning or extracting values from each column within a row.
- First version of SQLite released on 29/05/2000
- Latest version of SQLite is 3.31.1 released on 27/01/2020 (www.sqlite.org)

Creating Data Base Programmatically

- Create a data base helper class, say “DBAdapter”, first to avoid complexity and to make calling code easy to understand
- “DBAdapter” class, is inherited from the ***SQLiteOpenHelper*** class, which is an abstract class in Android to manage database creation and version management.
- In particular, you override the constructor, onCreate() and onUpgrade() methods of ***SQLiteOpenHelper*** class.

Creating Data Base Programmatically

1. Override the constructor of ***DBAdepter class.*** When the instance of DBAdepter class is created this constructor creates the database.
2. The **onCreate()** method creates a new database if the required database is not present.
3. The **onUpgrade()** method is called when the database needs to be upgraded.
4. You can then define the various methods for opening and closing the database, as well as the methods for adding/editing/deleting rows in the table.

Creating Data Base Programmatically

- The ***SQLiteDatabase*** class exposes ***insert***, ***update***, and ***delete*** methods that encapsulate the SQL statements required to perform these actions.
- The ***execSQL*** method lets you execute any valid SQL statement on your database tables.
- ***execSQL ("String")*** executes a single SQL statement that is ***NOT*** a SELECT or any other SQL statement that returns data.

Example of working with SQLite Database

1. Create helper class named ***“DBAdepter”*** **.** Then override constructor of this class. When the instance of DBAdepter class is created this constructor creates the database.

```
public class DBHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "elective.db";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_NAME = "students";
    private static final String col_1 = "id";
    private static final String col_2 = "name";
    private static final String col_3 = "email";

    public DBHelper(Context context)
    {
      super(context, DATABASE_NAME, factory:null, DATABASE_VERSION);
    }
```

Example of working with SQLite Database

2. The **onCreate()** method creates a new table if the required table is not present.

```
public void onCreate(SQLiteDatabase sqLiteDatabase)
    {
        sqLiteDatabase.execSQL("create table " +
    TABLE_NAME + "(id integer primary key autoincrement,
    name text, email text )");
    }
```

Example of working with SQLite Database

3. The **onUpgrade()** method is called when the database needs to be upgraded.

```
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1)
    {
        sqLiteDatabase.execSQL("drop table if exists "
    +TABLE_NAME);
        onCreate(sqLiteDatabase);
    }
```

onUpgrade method simply drops the existing table and replaces it with the new definition on upgrading the database version.

Example of working with SQLite Database

4. Inserting values into the table

```
public boolean insertvalues(String name, String email) {
        SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(col_2, name);
        contentValues.put(col_3, email);
        long result = sqLiteDatabase.insert(TABLE_NAME, null,
    contentValues);
        if (result == -1) return false;
        else return true;
    }
```

Example of working with SQLite Database

4. Inserting values into the table

- You use an object of ***ContentValues*** class to store key/value pairs.
- **put()** method of ***ContentValues*** class enables you to insert keys with values of different data types.
- Insert the new row by passing the ***ContentValues*** into the ***insert*** method called on the target database.

Example of working with SQLite Database

5. Extracting Values from Cursor

```
public Cursor getalldata()
    {
        SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
        Cursor result = sqLiteDatabase.rawQuery("select * from " +
    TABLE_NAME,null);
        return result;
    }
```

- Android uses the ***Cursor*** class as a return value for queries.
- Think of the Cursor as a pointer to the result set from a database query.

Example of working with SQLite Database

6. Updating the table values

```
public boolean updatedata(String id, String name, String email)
    {
        SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(col_1, id);
        contentValues.put(col_2, name);
        contentValues.put(col_3, email);
        sqLiteDatabase.update(TABLE_NAME,contentValues,"id = ?",  new String[]
    { id });
        return true;
    }
```

Update method requires four arguments: name of table, values to be updated, where, where arguments.

Example of working with SQLite Database

7. Deleting row from the table

```
public int deletedata(String id)
    {
       SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
       return sqLiteDatabase.delete(TABLE_NAME,"id = ?",new String[] { id });
    }
```

- To delete a row, simply call the ***delete*** method on a database, specifying the table name and a where clause that returns the rows you want to delete, as shown in the example.