Unit 4

1. **What is shared preference? Explain Creating, Saving, Restoring and Clearing the shared preference.**

Shared preferences allow you to read and write small amounts of primitive data as key/value pairs to a file on the device storage. The SharedPreference class provides APIs for getting a handle to a preference file and for reading, writing, and managing this data. The shared preferences file itself is managed by the framework and accessible to (shared with) all the components of your app. That data is not shared with or accessible to any other apps.

Creating a shared preferences file: You need only one shared preferences file for your app, and it is customarily named with the package name of your app. This makes its name unique and easily associated with your app. You create the shared preferences file in the onCreate() method of your main activity and store it in a member variable.

```
private String sharedPrefFile = "com.example.android.hellosharedprefs";
mPreferences = getSharedPreferences(sharedPrefFile,MODE_PRIVATE);
```

Saving a shared preference file: You save preferences in the onPause() state of the activity lifecycle using the SharedPreferences.Editor interface.

1. Get a SharedPreferences.Editor. The editor takes care of all the file operations for you. When two editors are modifying preferences at the same time, the last one to call apply wins.
2. Add key/value pairs to the editor using the put method appropriate for the data type. The put methods will overwrite previously existing values of an existing key.
3. Call apply() to write out your changes. The apply() method saves the preferences asynchronously, off of the UI thread. The shared preferences editor also has a commit() method to synchronously save the preferences. The commit() method is discouraged as it can block other operations

```
@Override
protected void onPause() {
    super.onPause();
    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.putInt("count", mCount);
    preferencesEditor.putInt("color", mCurrentColor);
    preferencesEditor.apply();
}
```

Restoring Shared Preferences: You restore shared preferences in the onCreate() method of your activity. The get() methods take two arguments—one for the key and one for the default value if the key cannot be found. Using the default argument, you don't have to test whether the preference exists in the file.

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
if (savedInstanceState != null) {
    mCount = mPreferences.getInt("count", 1);
    mShowCount.setText(String.format("%s", mCount));

    mCurrentColor = mPreferences.getInt("color", mCurrentColor);
    mShowCount.setBackgroundColor(mCurrentColor);
} else { … }
```

Clearing Shared Preferences: To clear all the values in the shared preferences file, call the clear() method on the shared preferences editor and apply the changes. You can combine calls to put and clear. However, when applying the preferences, the clear is always done first, regardless of whether you called clear before or after the put methods on this editor.

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.putInt("number", 42);
preferencesEditor.clear();
preferencesEditor.apply();
```

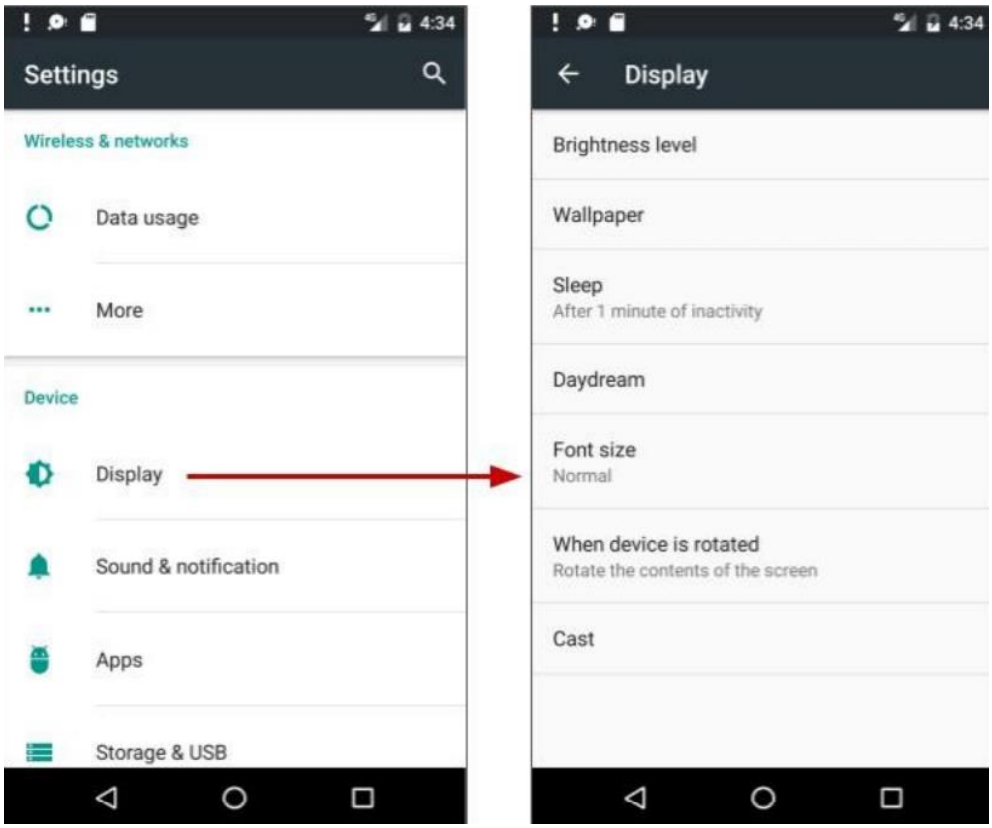
2. **User created a Login form. Once the user enters their details, the data will not be removed by unknowingly closing the app. What trick did he use in it and describe it? (Shared Preferences)**
3. **Describe the design guidelines for the Settings UI**

Settings should be well-organized, predictable, and contain a manageable number of options. A user should be able to quickly understand all available settings and their current values. Follow these design guidelines:

7 or fewer settings: Arrange them according to priority with the most important ones at the top.

7-15 settings: Group related settings under section dividers. For example, in the figure below, "Priority interruptions" and "Downtime (priority interruptions only)" are section dividers.

16 or more settings: Group related settings into separate sub-screens. Use headings, such as Display on the main Settings screen (as shown on the left side of the figure below) to enable users to navigate to the display settings (shown on the right side of the figure below):



4. Discuss the differences between Internal and External storage

Internal storage	External storage
Always available.	Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable.	World-readable. Any app can read.
When the user uninstalls your app, the system removes all your app's files from internal storage.	When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

5. Write a code to check read and write permission of external storage

To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file. This implicitly includes permission to read.

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

If your app needs to read the external storage (but not write to it), then you will need to declare the READ_EXTERNAL_STORAGE permission.

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

6. **Discuss the conditions for automatic backup of Android 6.0 and higher**

For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically backup app data to the cloud. The system performs this automatic backup for nearly all app data by default, and does so without your having to write any additional app code. When a user installs your app on a new device, or reinstalls your app on one (for example, after a factory reset), the system automatically restores the app data from the cloud. The automatic backup feature preserves the data your app creates on a user device by uploading it to the user's Google Drive account and encrypting it. There is no charge to you or the user for data storage, and the saved data does not count towards the user's personal Google Drive quota. Each app can store up to 25MB. Once its backed-up data reaches 25MB, the app no longer sends data to the cloud. If the system performs a data restore, it uses the last data snapshot that the app had sent to the cloud.

Automatic backups occur when the following conditions are met:

- The device is idle.
- The device is charging.
- The device is connected to a Wi-Fi network.
- At least 24 hours have elapsed since the last backup.

You can customize and configure auto backup for your app. See Configuring Auto Backup for Apps.

7. **How to check whether external storage is mounted?**

Always check whether external storage is mounted Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it. You can query the external storage state by calling getExternalStorageState(). If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files. For example, the following methods are useful to determine the storage availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

8. **What is a transaction? Describe the properties of a transaction**

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction. All changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by a program crash, an operating system crash, or a power failure.

Examples of transactions:

- Transferring money from a savings account to a checking account.
- Entering a term and definition into dictionary.
- Committing a changelist to the master branch.

ACID

- **Atomicity.** Either all of its data modifications are performed, or none of them are performed.
- **Consistency.** When completed, a transaction must leave all data in a consistent state.
- **Isolation.** Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state.
- **Durability.** After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

9. **Discuss the steps to implement an SQLite database with an example.**

To implement a database for your Android app, you need to do the following.

1. (Recommended) Create a data model.
2. Subclass SQLiteOpenHelper
    i. Use constants for table names and database creation query
    ii. Implement onCreate to create the SQLiteDatabase with tables for your data
    iii. Implement onUpgrade()
    iv. Implement optional methods
3. Implement the query(), insert(), delete(), update(), count() methods in SQLiteOpenHelper.
4. In your MainActivity, create an instance of SQLiteOpenHelper.
5. Call methods of SQLiteOpenHelper to work with your database.

Caveats:

- When you implement the methods, always put database operations into try/catch blocks.
- The sample apps do not validate the user data. When you write an app for publication, always make sure user data is what you expect to avoid the injection of bad data or execution of malicious SQL commands into your database.

Data model

It is a good practice to create a class that represents your data with getters and setters.

For an SQLite database, an instance of this class could represent one record, and for a simple database, one row in a table.

```
public class WordItem {
    private int mId;
    private String mWord;
    private String mDefinition;
    // Getters and setters and more
}
```

Subclass SQLiteOpenHelper

Any open helper you create must extend SQLiteOpenHelper.

```
public class WordListOpenHelper extends SQLiteOpenHelper {

    public WordListOpenHelper(Context context) {
       super(context, DATABASE_NAME, null, DATABASE_VERSION);
       Log.d(TAG, "Construct WordListOpenHelper");
    }
}
```

Define constants for table names While not required, it is customary to declare your table, column, and row names as constants. This makes your code a lot more readable, makes it easier to change names, and your queries will end up looking a lot more like SQL. You can do this in the open helper class, or in a separate public class; you will learn more about this in the chapter about content providers.

```
  private static final int DATABASE_VERSION = 1;
// has to be 1 first time or app will crash
private static final String WORD_LIST_TABLE = "word_entries";
private static final String DATABASE_NAME = "wordlist";

// Column names...
public static final String KEY_ID = "_id";
public static final String KEY_WORD = "word";

// ... and a string array of columns.
private static final String[] COLUMNS = {KEY_ID, KEY_WORD};
```

Define query for creating database

You need a query that creates a table to create a database. This is also customarily defined as a string constant. This basic example creates one table with a column for an auto-incrementing id and a column to hold words.

```
    private static final String WORD_LIST_TABLE_CREATE =
            "CREATE TABLE " + WORD_LIST_TABLE + " (" +
             KEY_ID + " INTEGER PRIMARY KEY, " +
             // will auto-increment if no value passed
             KEY_WORD + " TEXT );";
```

Implement onCreate() and create the database

The onCreate method is only called if there is no database. Create your tables in the method, and optionally add initial data.

```
@Override
public void onCreate(SQLiteDatabase db) { // Creates new database
   db.execSQL(WORD_LIST_TABLE_CREATE); // Create the tables
   fillDatabaseWithData(db); // Add initial data
   // Cannot initialize mWritableDB and mReadableDB here, because
   // this creates an infinite loop of on Create()
   // being repeatedly called.
}
```

Implement onUpgrade()

This is a required method.

If your database acts only as a cache for data that is also stored online, you can drop the the tables and recreate them after the upgrade is complete.

Note: If your database is the main storage, you **must** preserve the user's data before you do this as this operation destroys all the data. See the chapter on Storing Data.

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // SAVE USER DATA FIRST!!!
        Log.w(WordListOpenHelper.class.getName(),
                "Upgrading database from version " + oldVersion + " to "
                        + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + WORD_LIST_TABLE);
        onCreate(db);
}
```

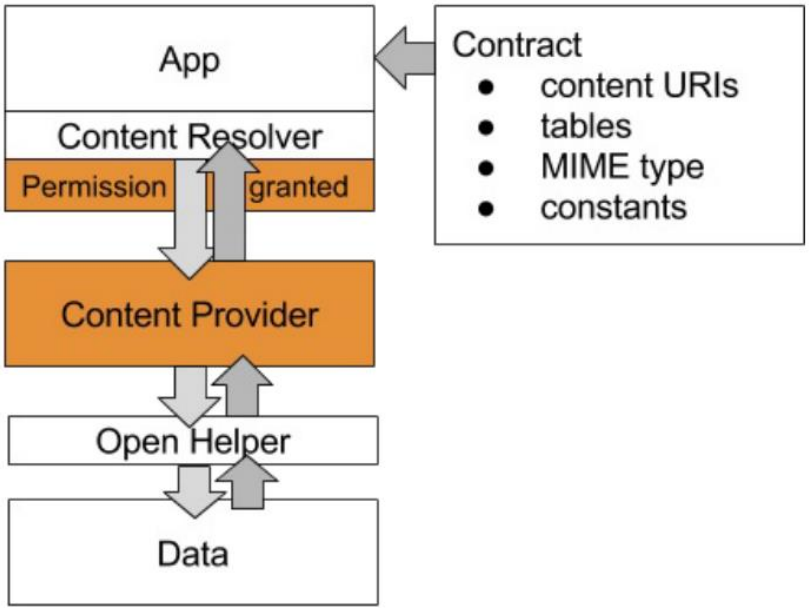
10. What is a Content Provider? Explain the App Architecture with a Content Provider.

A ContentProvider is a component that interacts with a repository. The app doesn't need to know where or how the data is stored, formatted, or accessed.

A content provider:

- Separates data from the app interface code
- Provides a standard way of accessing the data
- Makes it possible for apps to share data with other apps
- Is agnostic to the repository, which could by a database, a file system, or the cloud.

Architecturally, the content provider is a layer between the content-providing app's data storage backend and the rest of the app, separating the data and the interface.



Data and Open Helper: The data repository. The data could be in a database, a file, on the internet, generated dynamically, or even a mix of these. For example, if you had a dictionary app, the base dictionary could be stored in a SQLite database on the user's device. If a definition is not in the database, it could get fetched from the internet, and if that fails, too, the app could ask the user to provide a definition or check their spelling.

Data used with content providers is commonly stored in SQLite databases, and the content provider API mirrors this assumption.

Contract: The contract is a public class that exposes important information about the content provider to other apps. This usually includes the URI schemes, important constants, and the structure of the data that will be returned. For example, for the hat inventory app, the contract could expose the names of the columns that contain the price and name of a product, and the URI for retrieving an inventory item by part number.

Content Provider: The content provider extends the ContentProvider class and provides query(), insert(), update(), and delete() methods for accessing the data. In addition, it provides a public and secure interface to the data, so that other apps can access the data with the appropriate permissions. For example, to get inventory information from your app's database, the retail hat app would connect to the content provider, not to the database directly, as that is not permitted.

The app that owns the data specifies what permissions other apps need to have to work with the content provider. For example, if you have an app that provides inventory to retail stores, your app owns the data and determines the access permissions of other apps to the data. Permissions are specified in the Android Manifest.

Content Resolver: Content providers are always accessed through a content resolver. Think of the content resolver as a helper class that manages all the details of connecting to a content provider for you. Mirroring the content provider's API, the ContentResolver object provides you with query(), insert(), update(), and delete() methods for accessing data of a content provider. For example, to get get all the inventory items that are red hats, a hat store app would build a query for red hats, and use a content resolver to send that query to the content provider.

11. **User created a loader that works with content providers and databases. Suggest him a loader and help him how to implement it with an example.**
12. **Explain the steps required to implement a cursorloader with examples.**
13. **Describe the method to build URIs to access Content provider’s data**
14. **Discuss the general format of MIME type and its parts**