

Data Storage

Data Storage

❑ Following are the Data Storage options supported by Android

■ 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.

■ Network Connection

- Store data on the web with your own network server.

❑ Android provides a way for you to expose even your private data to other applications — with a content provider.

Using Shared Preferences

- ❑ Android tracks preferences for both users and the application, it does not differentiate between the two.
- ❑ The SharedPreferences class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types that can be shared between application components running in the same Context.
- ❑ This data will persist across user sessions (even if your application is killed).

Using Shared Preferences

- ❑ To get a [SharedPreferences](#) object for your application, use one of two methods:
 - [getSharedPreferences\(\)](#)
 - [getPreferences\(\)](#)
- ❑ Shared preferences are most commonly used to persist data across user sessions and to share settings between application components.

Using Shared Preferences

❑ To write values:

1. Call edit() to get a SharedPreferences.Editor.
2. Add values with methods such as putBoolean() and putString().
3. Commit the new values with commit()

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

Shared Preferences - Example

```
public class Calc extends Activity {  
    public static final String PREFS_NAME = "MyPrefsFile";  
    @Override  
    protected void onCreate(Bundle state){  
        super.onCreate(state);  
        int mode = Activity.MODE_PRIVATE;  
        ...  
        // Restore preferences  
        SharedPreferences settings = getSharedPreferences(PREFS_NAME, mode);  
        boolean silent = settings.getBoolean("silentMode", false);  
        setSilent(silent);  
    }  
}
```

// Continued ...

Shared Preferences - Example

```
@Override
protected void onStop(){
    super.onStop();
    // We need an Editor object to make preference changes.
    // All objects are from android.context.Context
    SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
    SharedPreferences.Editor editor = settings.edit();
    editor.putBoolean("silentMode", mSilentMode);

    // Commit the edits!
    editor.commit();
}
}
```

Using Internal Storage

- ❑ You can 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.
- ❑ 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\(\)](#).

Writing to Internal Storage - Example

```
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.

Other modes available

are: MODE_APPEND, MODE_WORLD_READABLE,
and MODE_WORLD_WRITEABLE.

Reading from Internal Storage


❑ To read a file from internal storage:

1. Call [openFileInput\(\)](#) and pass it the name of the file to read.

This returns a [FileInputStream](#).

2. Read bytes from the file with [read\(\)](#).

3. 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. you cannot write to the original file.

Reading from Internal Storage - Example

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

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

Saving cache files

- ❑ Temporary data can be cached instead of storing it on persistent storage.
- ❑ Use `getCacheDir()` to open a `File` that represents the internal directory where your application should save temporary cache files.
- ❑ When the device is low on internal storage space, Android may delete these cache files to recover space.
- ❑ You should maintain the files in cache instead of depending on Android to do that.
- ❑ Also the application should consume the memory in limit, around 1 Mb.

Useful Methods

❑ public abstract File getFilesDir ()

- Gets the absolute path to the file system directory where your internal files are saved.

❑ public abstract File getDir (String name, int mode)

- Creates (or opens an existing) directory within your internal storage space.

❑ public abstract boolean deleteFile (String name)

- Deletes a file saved on the internal storage.
- The name of the file to delete; can not contain path separators.

❑ public abstract String[] fileList ()

- Returns an array of files currently saved by your application.

External Storage

- ❑ Every Android-compatible device supports a shared "external storage" .
- ❑ Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

Checking media availability

- ❑ call [getExternalStorageState\(\)](#) to check whether the media is available.
- ❑ The media might be mounted to a computer, missing, read-only, or in some other state.

Checking External Media Availability - Example

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
    mExternalStorageAvailable = true;
    mExternalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all
    // we need
    // to know is we can neither read nor write
    mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```


Accessing files on external storage

- ❑ If you're using API Level 8 or greater, use `getExternalFilesDir()` to open a `File` that represents the external storage directory where you should save your files.
- ❑ This method takes a type parameter that specifies the type of subdirectory you want, such as `DIRECTORY_MUSIC` and `DIRECTORY_RINGTONES` (pass null to receive the root of your application's file directory).
- ❑ This method will create the appropriate directory if necessary.
- ❑ By specifying the type of directory, you ensure that the Android's media scanner will properly categorize your files in the system.
- ❑ When the application is uninstalled, this directory and all its contents will be deleted.

Accessing files on external storage

- ❑ If you're using API Level 7 or lower, use [getExternalStorageDirectory\(\)](#), to open a [File](#) representing the root of the external storage.
- ❑ You should then write your data in the following directory:
`/Android/data/<package_name>/files/`

Saving files that should be shared

- ❑ In API Level 8 or greater, use [getExternalStoragePublicDirectory\(\)](#), passing it the type of public directory you want, such as [DIRECTORY_MUSIC](#), [DIRECTORY_PICTURES](#), [DIRECTORY_RINGTONES](#), or others. This method will create the appropriate directory if necessary.

Saving files that should be shared

- ☐ 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.

Using Databases

- ❑ Android provides full support for SQLite database.
- ❑ All Android databases are stored in the `/data/data/<package_name>/databases` folder on your device (or emulator).
- ❑ All databases are private, accessible only by the application that created them.
- ❑ To share a database across applications, use Content Providers.

Introducing SQLite

- ❑ *SQLite is a relational database management system (RDBMS). It is:*
 - Open source
 - Standards-compliant
 - Lightweight
 - Single-tier
- ❑ It has been implemented as a compact C library that's included as part of the Android software stack.
- ❑ By providing functionality through a library, rather than as a separate process, each database becomes an integrated part of the application that created it.
- ❑ This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.

Cursors and Content Values

- ❑ ContentValues objects are used to insert new rows into database tables (and Content Providers).
- ❑ Each Content Values object represents a single row, as a map of column names to values.
- ❑ Queries in Android are returned as Cursor objects.
- ❑ The Cursor class includes several functions
 - **moveToFirst** Moves the cursor to the first row in the query result.
 - **moveToNext** Moves the cursor to the next row.
 - **moveToPrevious** Moves the cursor to the previous row.
 - **getCount** Returns the number of rows in the result set.

Cursors and Content Values

- **getColumnIndexOrThrow** Returns an index for the column with the specified name (throwing an exception if no column exists with that name).
- **getColumnName** Returns the name of the specified column index.
- **getColumnNames** Returns a String array of all the column names in the current cursor.
- **moveToPosition** Moves the cursor to the specified row.
- **getPosition** Returns the current cursor position.

Using the SQLiteOpenHelper

- ❑ SQLiteOpenHelper is an abstract class that wraps up the best practice pattern for creating, opening, and upgrading databases.
- ❑ By implementing and using an SQLiteOpenHelper, you hide the logic used to decide if a database needs to be created or upgraded before it's opened.
- ❑ Call `getReadableDatabase` or `getWritableDatabase` to open and return a readable/writable instance of the database.

Using the SQLiteOpenHelper – Example

```
dbHelper = new myDbHelper(context, DATABASE_NAME, null,  
    DATABASE_VERSION);  
SQLiteDatabase db;  
try {  
    db = dbHelper.getWritableDatabase();  
}  
catch (SQLException ex){  
    db = dbHelper.getReadableDatabase();  
}
```

Note: If the database doesn't exist, the helper executes its onCreate handler. If the database version has changed, the onUpgrade handler will fire. In both cases, the get<read/write>ableDatabase call will return the existing, newly created, or upgraded database as appropriate.

Opening and Creating Databases without the SQLiteHelper

- ❑ You can create and open databases without using the SQLiteHelper class with the openOrCreateDatabase method on the application Context.

```
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "mainTable";
private static final String DATABASE_CREATE =
    "create table " + DATABASE_TABLE +
    " ( _id integer primary key autoincrement," +
    "column_one text not null);";
SQLiteDatabase myDatabase;
private void createDatabase() {
    myDatabase = openOrCreateDatabase(DATABASE_NAME,
                                     Context.MODE_PRIVATE, null);

    myDatabase.execSQL(DATABASE_CREATE);
} http://javat.in
```

Querying Your Database

- ❑ All database queries are returned as a Cursor to a result set.
- ❑ To execute a query on a database, use the query method on the database object, passing in:
 - An optional **Boolean** that specifies if the result set should contain only unique values
 - The **name of the table** to query
 - A **projection, as an array of Strings**, that lists the columns to include in the result set
 - A **“where”** clause that defines the rows to be returned. You can include ? wildcards that will be replaced by the values stored in the selection argument parameter.
 - An array of selection argument strings that will replace the ?'s in the “where” clause

Querying Your Database

- A “**group by**” clause that defines how the resulting rows will be grouped
- A “**having**” filter that defines which row groups to include if you specified a “group by” clause
- A **String** that describes the order of the returned rows
- An **optional String** that defines a limit to the returned rows

Querying Your Database - Example

// Return all rows for columns one and three, no duplicates

```
String[] result_columns = new String[] {KEY_ID, KEY_COL1, KEY_COL3};
```

```
Cursor allRows = myDatabase.query(true, DATABASE_TABLE, result_columns,  
                                null, null, null, null, null, null);
```

// Return all columns for rows where column 3 equals a set value

// and the rows are ordered by column 5.

```
String where = KEY_COL3 + "=" + requiredValue;
```

```
String order = KEY_COL5;
```

```
Cursor myResult = myDatabase.query(DATABASE_TABLE, null, where,  
                                null, null, null, order);
```

Extracting Results from a Cursor

- ❑ To extract actual values from a result Cursor,
 - i. first use the moveTo<location> method
 - ii. Use the type-safe get methods (passing in a column index) to return the value stored at the current row for the specified column

Extracting Results from a Cursor - Example

```
int GOLD_HOARDED_COLUMN = 2;
Cursor myGold = myDatabase.query("GoldHoard", null, null, null, null,
null, null);
float totalHoard = 0f;
// Make sure there is at least one row.
if (myGold.moveToFirst()) {
    // Iterate over each cursor.
    do {
        float hoard = myGold.getFloat(GOLD_HOARDED_COLUMN);
        totalHoard += hoard;
    } while(myGold.moveToNext());
}
float averageHoard = totalHoard / myGold.getCount();
```


Adding, Updating, and Removing Rows

- ❑ The SQLiteDatabase class exposes specialized insert, delete, and update methods to encapsulate the SQL statements required to perform these actions.
- ❑ Any time you modify the underlying database values, you should call refreshQuery on any Cursors that currently have a view on the table.

Inserting New Rows

- ❑ To create a new row, construct a ContentValues object, and use its put methods to supply values for each column.
- ❑ Insert the new row by passing the Content Values object into the insert method called on the target database object — along with the table name.

Inserting New Rows - Example

```
// Create a new row of values to insert.  
ContentValues newValues = new ContentValues();  
// Assign values for each row.  
newValues.put(COLUMN_NAME, newValue);  
[ ... Repeat for each column ... ]  
// Insert the row into your table  
myDatabase.insert(DATABASE_TABLE, null, newValues);
```

Updating a Row on the Database

❑ Updating rows is also done using Content Values.

```
// Define the updated row content.
```

```
ContentValues updatedValues = new ContentValues();
```

```
// Assign values for each row.
```

```
updatedValues.put(COLUMN_NAME, newValue);
```

```
[ ... Repeat for each column ... ]
```

```
String where = KEY_ID + "=" + rowId;
```

```
// Update the row with the specified index with the new values.
```

```
myDatabase.update(DATABASE_TABLE, updatedValues, where, null);
```

Deleting Rows

- ❑ To delete a row, simply call delete on your database object, specifying the table name and a where clause that returns the rows you want to delete, as shown in the code below:

```
myDatabase.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```