

SAVING DATA

COMP 41690

DAVID COYLE

>

D.COYLE@UCD.IE

SAVING DATA

1. Saving key-value pairs of simple data types in a shared preferences file
2. Saving files in Android's file system
3. Saving data in SQL Databases

Materials: <https://developer.android.com/training/basics/data-storage/index.html>

SHARED PREFERENCES

For relatively small collections of key-values you should use the **SharedPreferences** APIs.

A **SharedPreferences** object points to a file containing key-value pairs and provides simple methods to read and write them.


SharedPreferences files are managed by the framework and can be private or shared.

ACCESSING A SHARED PREFERENCE FILE


You can create a new shared preference file or access an existing one by calling one of two methods:

1. **getSharedPreferences()** — Use this if you need multiple shared preference files identified by name which you specify with the first parameter. You can call this from any Context in your app.

```
Context context = getActivity();  
SharedPreferences sharedPref = context.getSharedPreferences(  
    getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```



The shared preferences file is identified by a string resource. You should use a name that is uniquely identifiable to your app



The mode:
Private means it is only accessible
By your app.

ACCESSING A SHARED PREFERENCE FILE

You can create a new shared preference file or access an existing one by calling one of two methods:

2. **getPreferences()** — Use this from an Activity if you need to use only one shared preference file for the activity.

This retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

WRITING SHARED PREFERENCES

To write to a shared preferences file, create a `SharedPreferences.Editor` by calling `edit()` on your `SharedPreferences`.

Pass the keys and values you want to write with methods such as `putInt()` and `putString()`. Then call `commit()` to save the changes.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

READING SHARED PREFERENCES

To retrieve values from a shared preferences file, call methods such as `getInt()` and `getString()`, providing the key for the value you want, and optionally a default value to return if the key isn't present.

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);  
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);  
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), defaultValue);
```

See example: [DataManagementSharedPreference](#)

SAVING FILES

Android uses a **Files** API that's similar to disk-based file systems on other platforms.

<https://developer.android.com/reference/java/io/File.html>

It builds on the basics of the Linux file system and the standard file input/output APIs in [java.io](#).

A **File** object is suited to reading or writing large amounts of data in ***start-to-finish*** order without skipping around. E.g. image files.

INTERNAL OR EXTERNAL

Internal storage:

- It's always available.
- Files saved here are accessible by only your app by default.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- 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()`.

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.

PERMISSIONS

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

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this will change in a future release.

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>
```

You don't need any permissions to save files on the internal storage.

SAVING TO INTERNAL STORAGE

When saving a file to internal storage, you can acquire the appropriate directory as a **File** by calling `getFilesDir()`. This returns a **File** representing an internal directory for your app.

Then create a new file:

```
File file = new File(context.getFilesDir(), filename);
```

Or write a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(string.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

INTERNAL CACHE

getCacheDir() - returns a File representing an internal directory for your app's temporary cache files.

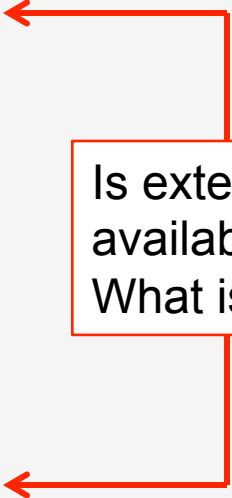
Be sure to delete each file once it is no longer needed.

```
public File getTempFile(Context context, String url) {  
    File file;  
    try {  
        String fileName = Uri.parse(url).getLastPathSegment();  
        file = File.createTempFile(fileName, null, context.getCacheDir());  
    } catch (IOException e) {  
        // Error while creating file  
    }  
    return file;  
}
```

This example extracts the file name from a URL and creates a file with that name in your app's internal cache directory

EXTERNAL STORAGE

```
/* 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;  
}
```



Is external storage
available?
What is it's state?

EXTERNAL STORAGE

1. Public files

Files that are freely available to other apps and to the user.

E.g. photos captured by your app or other downloaded files.

When the user uninstalls your app, these files remain available to the user.

```
public File getAlbumStorageDir(String albumName) {  
    // Get the directory for the user's public pictures directory.  
    File file = new File(Environment.getExternalStoragePublicDirectory(  
        Environment.DIRECTORY_PICTURES), albumName);  
    if (!file.mkdirs()) {  
        Log.e(LOG_TAG, "Directory not created");  
    }  
    return file;  
}
```

`getExternalStoragePublicDirectory()` takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as **DIRECTORY_MUSIC** or **DIRECTORY_PICTURES**.

EXTERNAL STORAGE

2. Private files

Files that rightfully belong to your app and should be deleted by the system when the user uninstalls your app.

E.g. resources downloaded by your app or temporary media files.

These files are accessible by the user and other apps because they are on the external storage, but realistically don't provide value to the user outside your app.

```
public File getAlbumStorageDir(Context context, String albumName) {  
    // Get the directory for the app's private pictures directory.  
    File file = new File(context.getExternalFilesDir(  
        Environment.DIRECTORY_PICTURES), albumName);  
    if (!file.mkdirs()) {  
        Log.e(LOG_TAG, "Directory not created");  
    }  
    return file;  
}
```

Any directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

USING DATABASES

SQLite is an open-source SQL database that stores data in a text file on a device. Android comes with a built-in SQLite database implementation.

SQLite supports all the relational database features.

The APIs you'll need are in the `android.database.sqlite` package:

<https://developer.android.com/training/basics/data-storage/databases.html>

Android stores your database in private disk space that's associated with an application. Data is secure, because by default this area is not accessible to other applications.

For tutorials on SQLite see:

<http://www.tutorialspoint.com/sqlite/index.htm>

http://suptonuts.sourceforge.net/readme_sqlite_tutorial.html

android.database.sqlite

Interfaces

SQLiteCursorDriver	A driver for SQLiteCursors that is used to create them and gets notified by the cursors it creates on significant events in their lifetimes.
SQLiteDatabase.CursorFactory	Used to allow returning sub-classes of Cursor when calling query.
SQLiteTransactionListener	A listener for transaction events.

Classes

SQLiteClosable	An object created from a SQLiteDatabase that can be closed.
SQLiteCursor	A Cursor implementation that exposes results from a query on a SQLiteDatabase .
SQLiteDatabase	Exposes methods to manage a SQLite database.
SQLiteOpenHelper	A helper class to manage database creation and version management.
SQLiteProgram	A base class for compiled SQLite programs.
SQLiteQuery	Represents a query that reads the resulting rows into a SQLiteQuery .
SQLiteQueryBuilder	This is a convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
SQLiteStatement	Represents a statement that can be executed against a database.

SQLiteOpenHelper class

Use this class to obtain references to your database.

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {  
    // If you change the database schema, you must increment the database version.  
    public static final int DATABASE_VERSION = 1;  
    public static final String DATABASE_NAME = "FeedReader.db";  
  
    public FeedReaderDbHelper(Context context) {  
        super(context, DATABASE_NAME, null, DATABASE_VERSION);  
    }  
    public void onCreate(SQLiteDatabase db) {  
        db.execSQL(SQL_CREATE_ENTRIES);  
    }  
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
        // This database is only a cache for online data, so its upgrade policy is  
        // to simply to discard the data and start over  
        db.execSQL(SQL_DELETE_ENTRIES);  
        onCreate(db);  
    }  
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
        onUpgrade(db, oldVersion, newVersion);  
    }  
}
```

SQLiteOpenHelper class

To access your database, instantiate your subclass of `SQLiteOpenHelper`

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

When you use the `SQLiteOpenHelper` class the system performs the potentially long-running operations of creating and updating the database only when needed and not during app startup.

You just need to call `getWritableDatabase()` or `getReadableDatabase()`.

Note:

Because they can be long-running, be sure to call `getWritableDatabase()` or `getReadableDatabase()` in a background thread, such as with `AsyncTask`.

PUT INFORMATION INTO A DATABASE

Insert data into the database by passing a `ContentValues` object to the `insert()` method.

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
    FeedEntry.TABLE_NAME,
    FeedEntry.COLUMN_NAME_NULLABLE,
    values);
```

READ INFORMATION

To read from a database, use the `query()` method, passing it your selection criteria and desired columns. The results of the query are returned to you in a `Cursor` object.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
};

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME, // The table to query
    projection,            // The columns to return
    selection,             // The columns for the WHERE clause
    selectionArgs,         // The values for the WHERE clause
    null,                 // don't group the rows
    null,                 // don't filter by row groups
    sortOrder,            // The sort order
);
```

READ INFORMATION

Generally, you should start by calling `moveToFirst()`, which places the "read position" on the first entry in the results.

```
cursor.moveToFirst();  
long itemId = cursor.getLong(  
    cursor.getColumnIndexOrThrow(FeedEntry._ID)  
);
```

For each row, you can read a column's value by calling one of the Cursor get methods, such as `getString()` or `getLong()`.

For each of the get methods, you must pass the index position of the column you desire, which you can get by calling `getColumnIndex()` or `getColumnIndexOrThrow()`.

DELETE INFORMATION

To delete rows from a table, you need to provide selection criteria that identify the rows.

The database API provides a mechanism for creating selection criteria that protects against SQL injection.

The mechanism divides the selection specification into a ***selection clause*** and ***selection arguments***.

Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

```
// Define 'where' part of query.  
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";  
// Specify arguments in placeholder order.  
String[] selectionArgs = { String.valueOf(rowId) };  
// Issue SQL statement.  
db.delete(table_name, selection, selectionArgs);
```


UPDATE A DATABASE

When you need to modify a subset of your database values, use the `update()` method.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

FURTHER DETAILS

For additional details and tutorials on databases in Android see:

<http://www.vogella.com/tutorials/AndroidSQLite/article.html>

<http://www.androidhive.info/2011/11/android-sqlite-database-tutorial/>

<https://www.sqlite.org/>