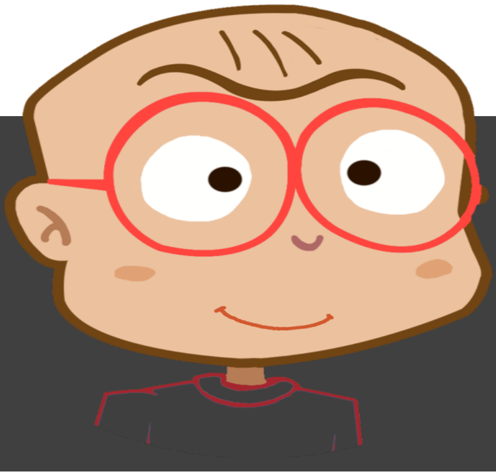


Saving Data

START »

Learning Goals

After the course, attendees will be able to:

- Have basic understanding about Android's data storage
- Have basic understanding about the ways to save data
- Have basic ability to create Android apps with different saving data methods

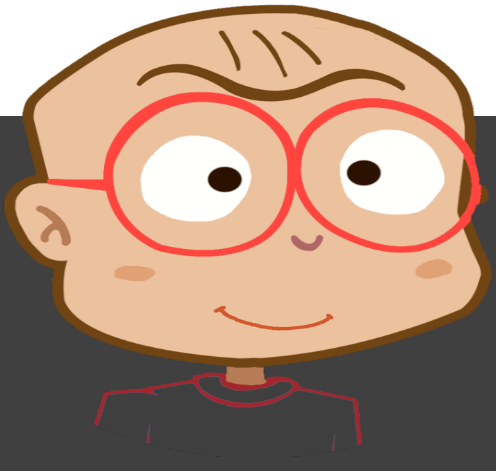
Table of contents

Overview

- Most Android apps need to save data such as information about the app state, user settings or large amounts of information in files and databases.
- The principal data storage options in Android:
  - Saving key-value pairs of simple data types in a shared preferences file
  - Saving arbitrary files in Android's file system
  - Using databases managed by SQLite

Overview

- Saving Key-Value Sets
  - Learn to use a shared preferences file for storing small amounts of information in key-value pairs.
- Saving Files
  - Learn to save a basic file, such as to store long sequences of data that are generally read in order.
- Saving Data in SQL Databases
  - Learn to use a SQLite database to read and write structured data.



Saving Key-Value Sets

START »

Saving Key-Value Sets

The **SharedPreferences** class provides a general framework that allows you to:

- Save and retrieve persistent key-value pairs of primitive data types.
- Save any primitive data: booleans, floats, ints, longs, and strings.
  - This data will persist across user sessions (even if your application is killed).

Saving Key-Value Sets

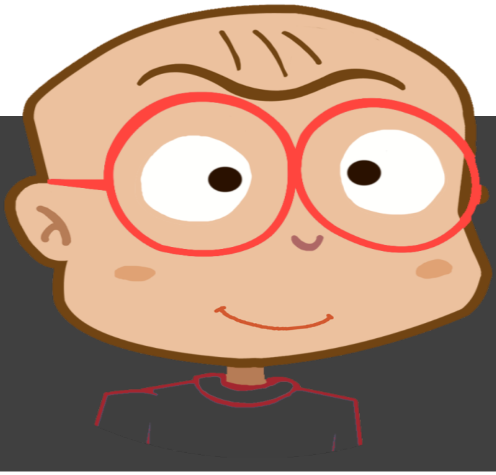
- Get a Handle to a SharedPreferences:
  - **getSharedPreferences**() — Use this if you need multiple shared preference files identified by name, which you specify with the first parameter.
  - **getPreferences**() — Use this from an Activity if you need to use only one shared preference file for the activity.
  - **getDefaultSharedPreferences**()?

Saving Key-Value Sets

- Write to Shared Preferences:
  - 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, 0);
    - SharedPreferences.Editor editor = settings.**edit()**;
    - editor.**putBoolean**("silentMode", mSilentMode);
    - editor.commit();

Saving Key-Value Sets

- Read from 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 settings = **getSharedPreferences**(PREFS_NAME, 0);
    - boolean silent = settings.**getBoolean**("silentMode", false);



Saving Files

START »

Saving Files

- Android uses a file system that's similar to disk-based file systems on other platforms.
- A File object is suited to reading or writing large amounts of data in start-to-finish order without skipping around.
- This lesson shows how to perform basic file-related tasks in your app. The lesson assumes that you are familiar with the basics of the Linux file system and the standard file input/output APIs in java.io.

Choose Internal or External Storage

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 it as USB storage or 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.

Save a File on Internal Storage

You can acquire the appropriate directory as a File by calling one of two methods:

- **getFilesDir()**

  Returns a File representing an internal directory for your app.

- **getCacheDir()**

  Returns a File representing an internal directory for your app's temporary cache files.

Save a File on Internal Storage

- To create a new file in one of these directories, you can use the File() constructor:
  - File file = new File(context.**getFilesDir()**, filename);
- Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory:
  - outputStream = **openFileOutput**(filename, Context.MODE_PRIVATE);
  - outputStream.write(string.getBytes());
  - outputStream.close();
- Or, if you need to cache some files, you should instead use createTempFile():
  - String fileName = Uri.parse(url).getLastPathSegment();
  - file = File.createTempFile(fileName, null, context.**getCacheDir()**);

Save a File on Internal Storage

- Other useful methods
  - **getDir()**
    - Creates (or opens an existing) directory within your internal storage space.
  - **deleteFile()**
    - Deletes a file saved on the internal storage.
  - **fileList()**
    - Returns an array of files currently saved by your application.

Save a File on External Storage

- Getting access to external storage:
  - acquire the **READ_EXTERNAL_STORAGE** or **WRITE_EXTERNAL_STORAGE** system permissions.

    -
      ```
      <manifest ...>
          <uses-permission
      android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
          ...
      </manifest>
      ```
  - These permissions are not required if you're reading or writing only files that are private to your app (4.4 & up).

Save a File on External Storage

- Checking media availability:
  - You should always call **getExternalStorageState()** to check whether the media is available.
    - 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.

```
/* 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;
}
```

Save a File on External Storage

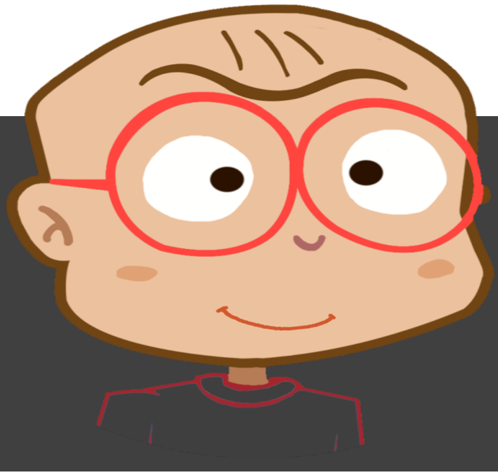
- Saving files that can be shared with other apps:
  - Call **getExternalStoragePublicDirectory()**, passing it the type of directory you want, such as DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_RINGTONES, or others.
    - // Get the directory for the user's public pictures directory.
      File file = new File(Environment.**getExternalStoragePublicDirectory**(
      Environment.DIRECTORY_PICTURES), albumName);
  - By doing this, the system's media scanner can properly categorize your files in the system
    - How to prevent this? (Hiding your files from the Media Scanner)
    - Include an empty file named **.nomedia** in your external files directory.

Save a File on External Storage

- Saving files that are app-private:
  - Use a private storage directory on the external storage by calling **getExternalFilesDir()**.
    - // Get the directory for the app's private pictures directory.
      File file = new File(context.**getExternalFilesDir**(
      Environment.DIRECTORY_PICTURES), albumName);
  - Remember that **getExternalFilesDir()** creates a directory inside a directory that is deleted when the user uninstalls your app.
  - How about the case that: the device has one more SD card slot?
    - You should use: **getExternalFilesDirs()**

Save a File on External Storage

- Saving cache files:
  - To open a File that represents the external storage directory where you should save cache files, call **getExternalCacheDir()**.
  - If the user uninstalls your application, these files will be automatically deleted.
  - Similar to **ContextCompat.getExternalFilesDirs()**, mentioned above, you can also access a cache directory on a secondary external storage (if available) by calling **ContextCompat.getExternalCacheDirs()**.



Saving Data in SQL Databases

START »

Saving Data in SQL Databases

- Android default Database engine is Lite. SQLite is a lightweight transactional database engine that occupies a small amount of disk storage and memory.
- Things to consider when dealing with SQLite:
  - Data type integrity is not maintained in SQLite, you can put a value of a certain data type in a column of another datatype (put string in an integer and vice versa).
  - Referential integrity is not maintained in SQLite, there is no FOREIGN KEY constraints or JOIN statements.
  - SQLite Full Unicode support is optional and not installed by default.

Saving Data in SQL Databases

- Create a Database Using a SQL Helper:
  - The first step is to create a class that inherits from SQLiteOpenHelper class. This class provides two methods to override to deal with the database:
    - onCreate(SQLiteDatabase db): invoked when the database is created, this is where we can create tables and columns to them, create views or triggers.
    - onUpgrade(SQLiteDatabse db, int oldVersion, int newVersion): invoked when we make a modification to the database such as altering, dropping , creating new tables.

Saving Data in SQL Databases

- Executing SQL Statements:
  - Execute any SQL statement : insert, delete, update or DDL using db.execSQL(String statement)
  - db.execSQL("CREATE TABLE " + **MyTable**.TABLE_NAME + " (“

        + **MyTable**._ID + " INTEGER PRIMARY KEY, “

        + **MyTable**.COLUMN_NAME_TITLE + " TEXT,“

        + **MyTable**.COLUMN_NAME_DESC + " TEXT)");
  - public static abstract class **MyTable** implements **BaseColumns** {
    - public static final String TABLE_NAME = "table_name";
    - public static final String COLUMN_NAME_TITLE = “col_name_title";
    - public static final String COLUMN_NAME_DESC = “col_name_desc“;
  - }

Saving Data in SQL Databases

- Put Information into a Database:
  - 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(**MyTable**.COLUMN_NAME_TITLE, content);
      values.put(**MyTable**.COLUMN_NAME_DESC, content);
  - Insert the new row, returning the primary key value of the new row.
    - long newRowId = db.insert(**MyTable**.TABLE_NAME, **MyTable**.COLUMN_NAME_NULLABLE, values)

Saving Data in SQL Databases

- Read Information from a Database:
  - To read from a database, use the query() method, passing it your selection criteria and desired columns:
    - String[] projection = { **MyTable**._ID,
      MyTable.COLUMN_NAME_TITLE,
      MyTable.COLUMN_NAME_DESC };
    - String sortOrder = **MyTable**.COLUMN_NAME_TITLE + " DESC";
    - Cursor c = db.query(

```
MyTable.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 );
```

Saving Data in SQL Databases

- Update a Database:
  - Updating the table combines the content values syntax of insert() with the where syntax of delete():
    - ContentValues values = new ContentValues();
      values.put(**MyTable**.COLUMN_NAME_TITLE, title);
      values.put(**MyTable**.COLUMN_NAME_DESC, desc);

    - String selection = **MyTable**._ID + " = ?";
    - String[] selectionArgs = { String.valueOf(rowId) };
    - int count = db.update(**MyTable**.TABLE_NAME, values, selection, selectionArgs);

Saving Data in SQL Databases

- Delete Information from a Database:
  - To delete rows from a table, you need to provide selection criteria that identify the rows:
    - // Define 'where' part of query.
      String selection = **MyTable**._ID + " = ?";
    - // Specify arguments in placeholder order.
      String[] selectionArgs = { String.valueOf(rowId) };
    - // Issue SQL statement.
      db.delete(**MyTable**.TABLE_NAME, selection, selectionArgs);

Saving Data in SQL Databases

- Managing Cursors:
  - Result sets of queries are returned in Cursor objects. There are some common methods that you will use with cursors:
    - boolean **moveToNext()**: Move the cursor to the next row. This method will return false if the cursor is already past the last entry in the result set.
    - boolean **moveToFirst()**: Move the cursor to the first row. This method will return false if the cursor is empty.
    - boolean **moveToPosition(int position)**: Move the cursor to an absolute position. The valid range of values is -1 <= position <= count. This method will return true if the request destination was reachable, otherwise, it returns false.
    - boolean **moveToPrevious()**: Move the cursor to the previous row. This method will return false if the cursor is already before the first entry in the result set.
    - boolean **moveToLast()**: Move the cursor to the last row. This method will return false if the cursor is empty.

Summary

Saving Key-Value Sets	Saving Files	Saving Data in SQL Databases
Learned to use a shared preferences file for storing small amounts of information in key-value pairs..	Learned to save a basic file, such as to store long sequences of data that are generally read in order.	Learned to use a SQLite database to read and write structured data.

Exit Course

THANK YOU

EXIT