ANDROID APPS DEVELOPMENT FOR MOBILE AND TABLET DEVICE (LEVEL II)

Lecture 2: Data Storage

Peter Lo

Overview

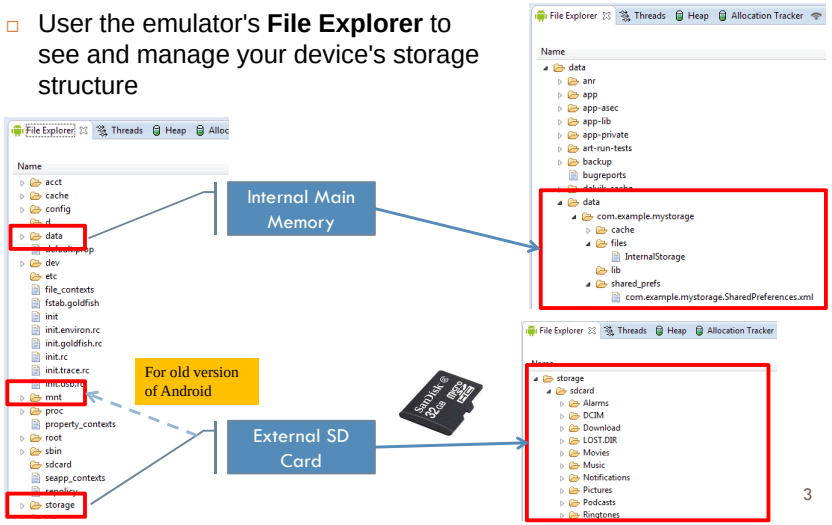
- Android provides several options for you to save persistent application data.

Storage Option	Description
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 (e.g. SD Card).
SQLite Databases	Store structured data in a private database.
Network Connection	Store data on the web with network server

The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications and how much space your data requires

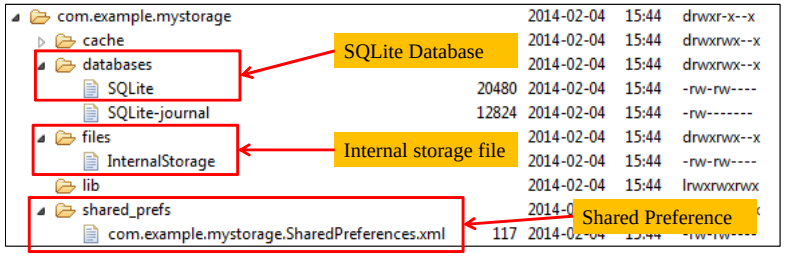
Android Folder Structure

- User the emulator's **File Explorer** to see and manage your device's storage structure



Android File Structure

- Android allows to persists application data via the file system.
- For each application the Android system creates a *data/data/[application package]* directory.
- On Android, all internal application data objects (including files) are private to that application.



Android Life Cycle

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

All activities must implement ***onCreate()*** to do the initial setup when the object is first instantiated

Activities should implement ***onPause()*** to commit data changes in anticipation to stop interacting with the user

Shared Preferences

- Is an Android lightweight mechanism to store and retrieve <Key-Value> pairs of primitive data types.
- If you have a relatively small collection of key-values that you'd like to save, you should use the Shared Preferences.
- It is typically used to keep state information and shared data among several activities of an application.
- In each entry of the form <key-value> the key is a string and the value must be a primitive data type.

key	value
firstName	Bugs
lastName	Bunny
location	Earth

Shared preferences are not strictly for saving "user preferences" such as what ringtone a user has chosen.

Using Shared Preferences

- Android supports the usage of the *SharedPreferences* class for persisting key-value pairs of primitive data types in the Android file system.
- The *SharedPreferences* class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types.
- The default preferences are available via the method *PreferenceManager.getDefaultSharedPreferences()*.

Get a Handle to a SharedPreferences

- There are two methods to access the preference:
  - getPreferences()
    - Use this from an Activity if you need to use only **one** shared preference file for the activity.

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

  - getSharedPreferences()
    - Use this if you need **multiple** shared preference files identified by name, which you specify with the first parameter.

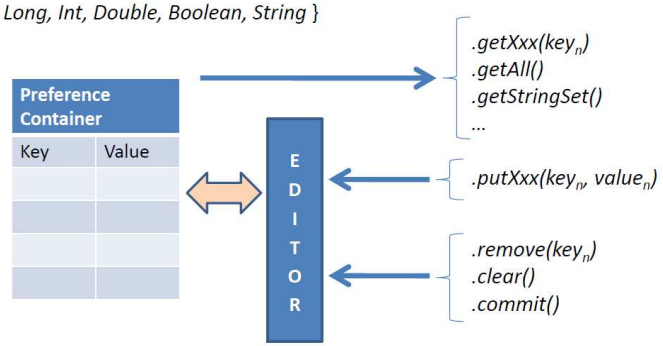
```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

When naming your shared preference files, you should use a name that's uniquely identifiable to your app, such as "com.example.myapp.PREFERENCE_FILE_KEY“

Using Preferences API calls

- All of the get preference methods return a preference object whose contents can be manipulated by an editor that allows put and get commands to place data in and out of the preference container.

Xxx = { Long, Int, Double, Boolean, String }



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.

```
// Retrieve and hold the contents of the preferences file
SharedPreferences preferences = getSharedPreferences(PREFERENCE_FILE, Context.MODE_PRIVATE);

// Retrieve a String value from the preferences
mDataString = preferences.getString(PREFERENCE_KEY, "");
```

Default value to return if the preference not exist

Operating mode
- Use 0 or MODE_PRIVATE for the default operation,
- MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE to control permissions.
- MODE_MULTI_PROCESS used if multiple processes are mutating the same SharedPreferences file.

Write to Shared Preferences

- To write to a shared preferences file, create a *SharedPreferences.Editor* by calling *edit()* on your Shared Preferences.
- Pass the keys and values you want to write with methods such as *putInt()* and *putString()*.
- Call *commit()* to save the changes.

```
// Retrieve and hold the contents of the preferences file
SharedPreferences preferences = getSharedPreferences(PREFERENCE_FILE, Context.MODE_PRIVATE);

// Create a new Editor for these preferences
SharedPreferences.Editor editor = preferences.edit();

// Set a String value in the preferences editor
editor.putString(PREFERENCE_KEY, mDataString);

// Commit your preferences changes
editor.commit();
```


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.
  - E.g. it's good for image files or anything exchanged over a network
- All Android devices have two file storage areas:
  - Internal Storage – Built-in non-volatile memory
  - External Storage – Removable storage medium (such as micro SD card)
- Some devices divide the permanent storage space into internal and external partitions, so even without a removable storage medium, there are always two storage spaces.

Internal Storage vs. External Storage

Internal Storage

- 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

- Not always available (e.g. USB storage)
- 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.

Using the Internal Storage

- You can save files directly on the device's internal storage.
- Files saved to the internal storage are private to your application and other applications cannot access them.
- When the user uninstalls your application, these files are removed.

Write a File to Internal Storage

- To create and write a private file to the internal storage:
  - Call *openFileOutput()* with the name of the file and the operating mode. This returns a *FileOutputStream*.
  - Write to the file with *write()*.
  - Close the stream with *close()*.

```
// Open a private file for writing
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);

// Constructs a new DataOutputStream on the OutputStream out
DataOutputStream out = new DataOutputStream(fos);

// Write to file in Unicode format
out.writeUTF(mDataString);

// Close file
out.close();

// Closes the file stream.
fos.close();
```

MODE_PRIVATE will create the file and make it private to your application.
Other modes available are: MODE_APPEND, MODE_WORLD_READABLE, and MODE_WORLD_WRITEABLE.

Read a File from Internal Storage

- To read a file from internal storage:
  - Call *openFileInput()* and pass it the name of the file to read. This returns a *FileInputStream*.
  - Read bytes from the file with *read()* or *readUTF()*.
  - Close the stream with *close()*.

```
// Open a private file for reading
FileInputStream fos = openFileInput(FILENAME);

// Constructs a new DataInputStream on the InputStream in
DataInputStream in = new DataInputStream(fos);

// Read the content in Unicode format
mDataString = in.readUTF();

// Closes this stream
in.close();

// Closes the file stream.
fos.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 (but you cannot write to the original file).

External Storage

- Android-compatible device supports a shared external storage that you can use to save files.
- This can be a removable storage media (such as an SD card) or an internal (non-removable) storage.
- It's possible that a device using a partition of the internal storage for the external storage may also offer an SD card slot. In this case, the extra storage is intended only for user-provided media that the system scans.

External storage can become unavailable if the user mounts the external storage on a computer or removes the media, and there's no security enforced upon files you save to the external storage. All applications can read and write files placed on the external storage and the user can remove them.

Checking Media Availability

- Before you do any work with the external storage, you should always check whether the media is available.
- The media might be mounted to a computer, missing, read-only, or in some other state.

```
/* 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;
}
```

The ***getExternalStorageState()*** method returns other states that you might want to check, such as whether the media is being shared (connected to a computer), is missing entirely, has been removed badly, etc. You can use these to notify the user with more information when your application needs to access the media.

Files Categories of External Storage

Public Directory

- Files that should be freely available to other apps and to the user.
- When the user uninstalls your app, these files should remain available to the user.
- E.g. photos captured by your app or other downloaded files.

Private Directory

- Files that rightfully belong to your app and should be deleted when the user uninstalls your app.
- Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app.
- When the user uninstalls your app, the system deletes all files in your app's external private directory.
- E.g. additional resources downloaded by your app or temporary media files.

Creating Files in Public Directory

- If you want to save public files on the external storage, use the *getExternalStoragePublicDirectory()* method to get a File representing the appropriate directory on the external storage.
- The method 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*.

```
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;
}
```

Permission for Access SD Card

- When you deal with external files you need to request permission to read and write on the SD card.
- The following clauses need to add to the AndroidManifest.xml

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

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

Creating Files in Private Directory

- If you want to save files that are private to your app, you can acquire the appropriate directory by calling *getExternalFilesDir()* and passing it a name indicating the type of directory you'd like.
- Each 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.

```
public File getExternalFilesDir(String FolderName) {
    // Get the directory for the private directory
    File folder = new File(Environment.getExternalStorageDirectory(), FolderName);

    // Create the folder
    if (!folder.mkdirs()) {
        Log.e("LOG_TAG", "Directory not created");
    }

    // Return the folder name
    return folder;
}
```


Hiding your files from the Media Scanner

- Include an empty file named ***.nomedia*** in your external files directory.
- This prevents media scanner from reading your media files and providing them to other apps through the MediaStore content provider.
- However, if your files are truly private to your app, you should save them in an app-private directory

SQLite

- Using an SQLite database in Android does not require a setup procedure or administration of the database.
  - SQLite is embedded into every Android device.
- You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.

create a database,	Insert rows,
define SQL tables,	delete rows,
indices,	change rows,
queries,	run queries and
views,	administer a SQLite database file
triggers	

Data Type Conversion

- SQLite only supports the following data types:
  - TEXT (similar to String in Java)
  - INTEGER (similar to long in Java)
  - REAL (similar to double in Java)
- All other types must be converted into one of these fields before getting saved in the database.
- SQLite itself does not validate if the types written to the columns are actually of the defined type
  - E.g. you can write an integer into a string column and vice versa.

Beware of Sharing Issues

- You cannot access internal databases belonging to other people (instead use Content Providers or external SD resident DBs).
- An SD resident database requires the declaration of following permissions in AndroidManifest file:

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

SQLITE (as well as most DBMS) is not case sensitive

Create a Database

- To create a database in your Android application you create a subclass of the *SQLiteOpenHelper* class and override the following methods to create and update your database.
  - *onCreate()* – is called by the framework, if the database is accessed but not yet created.
  - *onUpgrade()* – is called if the database version is increased in your application code. This method allows you to update an existing database schema or to drop the existing database and recreate it via the *onCreate()* method.
- The database tables should use the identifier *_id* for the primary key of the table. Several Android functions rely on this standard.

Create a Database Using a SQL Helper

```
public class MySQLiteOpenHelper extends SQLiteOpenHelper {
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "MyDatabase";

    public MySQLiteOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    public void onCreate(SQLiteDatabase db) {
        String SQL_CREATE_ENTRIES = "CREATE TABLE ...";
        db.execSQL(SQL_CREATE_ENTRIES);
    }

    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS ...");
        onCreate(db);
    }
}
```

If you change the database schema, you must increment the database version.

Called by the framework if the database not yet created

If the database version is increased in your application code. This method allows you to update an existing database schema or to drop the existing database and recreate it

SQLiteDatabase

- *SQLiteDatabase* provides the *insert()*, *update()*, *delete()* and *execSQL()* method, which allows to execute SQL statement directly.
- The object *ContentValues* allows to define key/values, can be used for inserts and updates of database entries.
  - The key represents the table column identifier
  - The value represents the content for the table record in this column.
- Queries can be created via the *rawQuery()* and *query()* methods or via the *SQLiteQueryBuilder* class .
  - *rawQuery()* directly accepts an SQL select statement as input.
  - *query()* provides a structured interface for specifying the SQL query.
  - *SQLiteQueryBuilder* helps to build SQL queries.

Put Information into a Database

- Insert data into the database by passing a *ContentValues* object to the *insert()* method:

```
SQLiteDatabase db = this.getWritableDatabase();

ContentValues values = new ContentValues();
values.put(COL_ID, id);
values.put(COL_NAME, value);

long newRowId = db.insert(TABLE_NAME, null, values);

db.close();
```

Get reference to writable database

Create a new map of values, where column names are the keys

Insert the new row, returning the primary key of the new row
- *table* – the table to insert the row into
- *nullColumnHack* – optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If not set to null, this parameter provides the name of nullable column name to explicitly insert a NULL into in the case where your values is empty.
- *values* – this map contains the initial column values for the row. The keys should be the column names and the values the column values

Close the database connection

Selection Criteria

- 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.
  - The clause defines the columns to look at, and also allows you to combine column tests.
  - The arguments are values to test against that are bound into the clause.
- Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

Delete Information from a Database

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

```
SQLiteDatabase db = this.getWritableDatabase();

String selection = COL_NAME + " LIKE ?";

String[] selectionArgs = { String.valueOf(rowId) };

db.delete(TABLE_NAME, selection, selectionArgs);

db.close();
```

Get reference to writable database

Define 'where' part of query. Passing "1" will remove all rows.

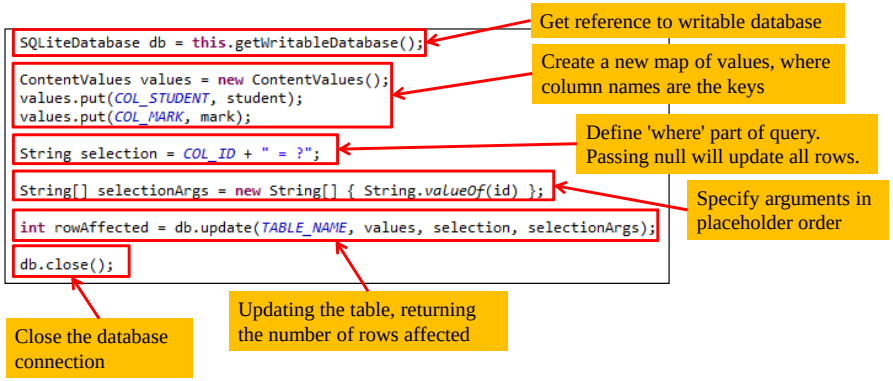
Specify arguments in placeholder order

Delete the row, returning the number of rows affected

Close the database connection

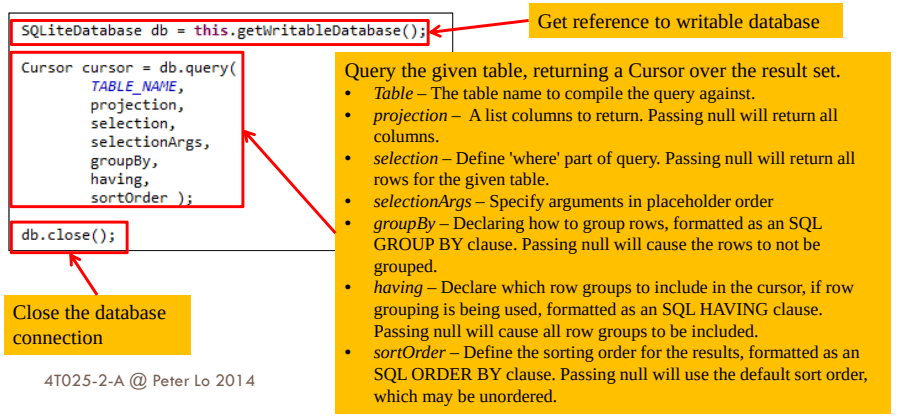
Update a Database

- When you need to modify a subset of your database values, use the *update()* method. Updating the table combines the content values syntax of *insert()* with the where syntax of *delete()*.



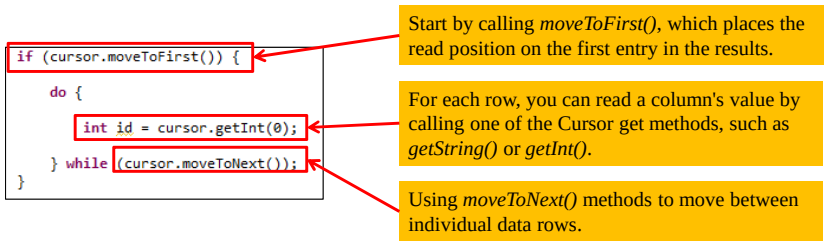
Read Information from a Database

- 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.



Using Cursor

- A Cursor represents the result of a query and basically points to one row of the query result. This way Android can buffer the query results efficiently; as it does not have to load all data into memory.
- To look at a row in the cursor, use one of the Cursor move methods, which you must always call before you begin reading values.



Common Cursor Method

- Generally, you should start by calling *moveToFirst()*, which places the read position on the first entry in the results, and *moveToNext()* methods to move between individual data rows.
- For each row, you can read a column's value by calling one of the Cursor get*() methods:
  - *getCount()* to get the number of elements.
  - *getLong()*, *getString()* to access the column data for the current position of the result.
  - *getColumnIndex()*, *getColumnIndexOrThrow()* method get the column index for a column name of the table.
- The *isAfterLast()* method allows to check if the end of the query result has been reached.
- A Cursor needs to be closed with the *close()* method call.