Android
IV –Persistent Data

Stefan BORNHOFEN
EISTI

Data storage

Android provides several options to save persistent application data.

Your data storage options are the following:

- **Traditional files**: Internal (private data on the device memory) or external (public data on shared external storage, e.g. SD card)
- **Shared Preferences**: Store private data in key-value pairs.
- **SQLite Databases**: Store structured data in a private database.
- **Network** : Store data on the web with your own network server.

Internal vs. External storage

- Android devices have two file storage areas: "internal" and "external" storage.
- These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage).

Internal storage:

- Always available.
- By default, files saved here are accessible by your app only (unless you specify otherwise)
- When the user uninstalls your app, the system removes all your app's files from internal storage.

External storage:

- 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()`.

Files

- 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
- When the user uninstalls your application, these files are removed.

Write data

```
String FILENAME = "file.txt";
String string = "hello world!";
FileOutputStream fos =
    openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

Files

- 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
- When the user uninstalls your application, these files are removed.

Read data

```
FileInputStream fis = context.openFileInput(fileName);
InputStreamReader isr = new InputStreamReader(fis);
StringBuilder sb = new StringBuilder();
char[] inputBuffer = new char[2048];
int l;
while ((l = isr.read(inputBuffer)) != -1) {
     sb.append(inputBuffer, 0, l);
}
String readString = sb.toString(); // bytes to string
fis.close();
```

Shared Preferences

- Simple way to share a small amount of data
- Private by default but can be shared with other applications
- Key-value pairs of simple data types
- boolean, float, int, long, and string
- XML file

Shared Preferences: example

```
SharedPreferences sp =
  PreferenceManager.getDefaultSharedPreferences(this);


// put data
SharedPreferences.Editor prefsEditor = sp.edit();
prefsEditor.putString("MY_NAME", "John");
prefsEditor.commit();



// get data
String name = sp.getString("MY_NAME", "no name");
```

SQLite Database

- Android provides full support for SQLite databases.
- Any databases you create will be accessible by name to any class in the application, but not outside the application.
- SQLite databases are stored in a single file.
- It can be stored in the internal memory or on the external SD card (default is internal memory: /data/data/<package name>/databases)
- Any Application can access an SD stored database. All that is needed is the path where the database file is located

Content Provider

- An SQLite database is **private** to the application which creates it. If you want to share data with other applications you can use a **ContentProvider**.
- A content provider is an optional component that exposes read/write access to application data, subject to whatever restrictions you want to impose.
- Android itself includes content providers that manage data such as audio, video, images, and personal contact information.

SQLite storage types

- NULL – null value
- INTEGER - signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value
- REAL - a floating point value, 8-byte IEEE floating point number.
- TEXT - text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- BLOB - the value is a blob of data, stored exactly as it was input.

class SQLiteDatabase

- Similar to JDBC
- Contains the methods for: creating, opening, closing, inserting, updating, deleting and quering an SQLite database

Create a Database

Create a subclass of SQLiteOpenHelper and override onCreate() :

```
public class MyHelper extends SQLiteOpenHelper {
  private static final int DATABASE_VERSION = 1;
  private static final String DATABASE_NAME = "MYBASE";

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

  @Override
  public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE MyTable (
            id INTEGER PRIMARY KEY,
            data TEXT ) ");
  }
}
```

Action queries

- Grab an instance of your SQLiteOpenHelper
- Every time you write to the database, call `getWritableDatabase()`
- This returns a SQLiteDatabase object that represents the database and provides methods for SQLite operations.
- When your app is destroyed, close database by calling `close()`

Action query: Insert

- long insert(String table, String nullColumnHack, ContentValues values)

```
ContentValues values = new  ContentValues();
values.put("firstname", "J.K.");
values.put("lastname", "Rowling");
long authorID =
  myDatabase.insert("table_author",null,values);
```

Action query: Update

- int update(String table, ContentValues values, String whereClause, String[ ] whereArgs)

```
ContentValues  values = new ContentValues();
values.put("title" , "Harry Potter");
myDatabase.update("table_book",
   values ,"id=?" , new String[]{"15"});
```

Action query: Delete

- int delete(String table, String whereClause, String[] whereArgs)

```
myDatabase.delete("table_book",
      "id=?", new String[]{"15"});
```

Transactions

- Contribute to maintain consistent data and prevent unwanted losses due to abnormal termination of execution.
- **atomicity** : all parts of a method are fused in an indivisible statement.
- Process action queries inside the protective frame of a database transaction: "**complete success or total failure**"

Transactions

- The transaction is defined between the methods: `beginTransaction` and `endTransaction`.
- Call `setTransactionSuccessful` to commit any changes.
- The absence of it provokes an implicit rollback.

```
SQLiteDatabase db = this.getWritableDatabase();
db.beginTransaction();
try {
  //perform your action query here ...
  db.setTransactionSuccessful(); //commit
}
catch(SQLiteException e) {
  //report problem
}
finally {
  db.endTransaction();
}
```

Select queries

- Get an instance of your SQLiteOpenHelper implementation
- To read from to the database, call `getReadableDatabase()`
- This returns a SQLiteDatabase object that represents the database and provides methods for SQLite operations.
- A select query returns a cursor which allows exploring the result set
- When your app is destroyed, close database by calling `close()`

SQL-select basics

SQL-select statements are based on the following components

Select field1, field2, … , fieldn
From table1, table2, … , tablen
Where (restriction-join-condition)
Order by fieldn1, …, fieldnm
Group by fieldm1, … , fieldmk
Having (group-condition)

The first two lines are mandatory, the rest is optional.

Raw select queries (version 1)

```
String mySQL= "select count(*)
 from mytable
 where id > 1
 and name = 'toto'";
Cursor c1 = db.rawQuery(mySQL, null);
```

Raw select queries (version 2)

```
String[] args= {"1", "toto"};
String mySQL= "select count(*)
 from mytable
 where id > ?
 and name = ?";
Cursor c1 = db.rawQuery(mySQL, args);
```

Simple select queries

- Simple queries use a **template** implicitly representing a condensed version of a typical (non-joining) SQL select statement.
- No explicit SQL statement is made.
- Simple queries can only retrieve data from a **single table**

The method's signature has a fixed sequence of seven arguments representing:

1.the table name,
2.the columns to be retrieved,
3.search condition (where-clause),
4.arguments for the where-clause,
5.the group-by clause,
6.having-clause, and
7.the order-by clause.

```
query (
        String     table,
        String[]    columns,
        String     selection,
        String[]    selectionArgs,
        String     groupBy,
        String     having,
        String     orderBy
)
```

Simple select queries

- **SELECT * FROM ABC**

```
Cursor c =
db.query("abc",null,null,null,null,null,null);
```

- **SELECT * FROM ABC WHERE ID=5**

```
Cursor c = db.query("abc",null, "id=?" , new
String[] {"5"},null,null,null);
```

- **SELECT title,id FROM BOOKS ORDER BY title ASC**

```
Cursor c = db.query("books",new String []
{"title","id"},null,null,null,null,"title
ASC");
```

Cursor

- Read-only access to tables produced by select statements
- Provides one row-at-the-time operations on a table
- Includes many operators, among them:
  - *Positional awareness*: isFirst, isLast
  - *Navigation*: moveToFirst, moveToLast, moveToNext
  - *Extraction*: getInt, getString, getFloat, getBlob, getDate
  - *Inspection*: getColumnName, getColumnNames, getColumnIndex, getColumnCount, getCount

Cursor

```
String [] col ={"id", "name", "phone"};

Cursor myCur= db.query("mytable", col,
  null, null, null, null, "id");
int idCol   = myCur.getColumnIndex("id");
int nameCol = myCur.getColumnIndex("name");
int phoneCol= myCur.getColumnIndex("phone");

while (myCur.moveToNext()) { // re-use col
  col[0] = Integer.toString((myCur.getInt(idCol)));
  col[1] = myCur.getString(nameCol);
  col[2] = myCur.getString(phoneCol);
  txtMsg.append("\n"+col[0]+" "+ col[1]+" "+col[2]);
}
```

ListView

```
public class MainActivity extends AppCompatActivity {
    ArrayAdapter<String> adapter;
    String[] items = {"A", "B", "C", "D", "E", "F", "G", "H"};
    ArrayList<String> list;
    ListView listView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayList list = new ArrayList<String>(Arrays.asList(items));
        adapter = new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, list);
        listView = (ListView) findViewById(R.id.listView);
        listView.setAdapter(adapter);
    }
}
```

Exercise

Write an interactive ToDo List (2p).
Enter text in dialog windows.

- Two actions: “edit name” and “add job”
- User name stored in the shared preferences
- Jobs stored in a SQLite database
- Long click deletes a job

Add the following feature (1p).
The user can dump the current jobs to a text file (“export”), and restore them at a later stage (“import”).