

INTERESTING CULTURAL ARTEFACTS	PERSISTANCE	FILESYSTEM	SQL STORAGE	DISCUSSION
<h1>Persistence</h1> <p>CE881: Mobile and Social Application Programming</p> <p>Spyros Samothrakis</p> <p>February 22, 2016</p>				
1 / 49				

INTERESTING CULTURAL ARTEFACTS	PERSISTANCE	FILESYSTEM	SQL STORAGE	DISCUSSION
<p>Interesting Cultural Artefacts</p> <p>Persistence</p> <p>Filesystem</p> <p>SQL storage</p> <p>Discussion</p>				
2 / 49				

INTERESTING CULTURAL ARTEFACTS	PERSISTANCE	FILESYSTEM	SQL STORAGE	DISCUSSION
<h2>THEME: “PERSISTENCE AND MEMORY”</h2> <ul style="list-style-type: none"><li>▶ Johny Mnemonic (Movie - 1995)</li><li>▶ 320GB of storage was a big thing</li><li>▶ Total Recall</li></ul>				
3 / 49				

INTERESTING CULTURAL ARTEFACTS	PERSISTANCE	FILESYSTEM	SQL STORAGE	DISCUSSION
<h2>PRESENTATIONS</h2> <ul style="list-style-type: none"><li>▶ Week 8 (i.e. next week)<ul style="list-style-type: none"><li>▶ No need to prepare anything</li><li>▶ Short, 5 minutes presentation on what you do</li></ul></li><li>▶ Week 10 (lab 10) short presentations/ questions</li><li>▶ Assiment 2 to be released shortly</li></ul>				
4 / 49				

## MAIN APPROACHES

- ▶ Saving Key-Value pairs in preferences
  - ▶ Easy but limited
- ▶ Using the File System
  - ▶ Android sits on top of a Linux file system
  - ▶ There are restrictions on where files can be opened (more of this later)
  - ▶ But once you have a `FileInputStream` or a `FileOutputStream`, it is simply standard Java
- ▶ Using an SQLite database
  - ▶ Lightweight standalone SQL database
  - ▶ Standard on Android Platform

5 / 49

## BINARY VERSUS CHARACTER DATA

- ▶ jpeg, mp3, Java serialized objects are all binary format
  - ▶ This means that each byte in the body of the file can contain any bytes
- ▶ Any one with the word Reader or Writer is limited to character data
  - ▶ Writing certain bytes will not work: they will be escaped and cause an error in the binary file format
  - ▶ Example: try writing a jpeg image using a `FileWriter` object;
    - ▶ It won't work!

6 / 49

## CHARACTER DATA

- ▶ Plain text files
- ▶ HTML, XML files
- ▶ But: sometimes we need to encode binary data over Character channels
- ▶ **Question:** when does this need arise?
  - ▶ Solution: use base-64
  - ▶ This encodes arbitrary byte sequences using bytes allowed character sequences
  - ▶ The cost is that it requires 1/3 more space
  - ▶ Each 3 bytes of binary require 4 bytes of base 64

7 / 49

## KEY-VALUE SETS

- ▶ Like Map structures in Java
- ▶ But limited in the range of values they can store
- ▶ Cannot store general Objects
- ▶ Writing:

```

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

8 / 49

## READING FROM A KEY-VALUE SET

```
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);
```

Note: can only save simple types and Strings

## PLAIN TEXT FILES

- ▶ HTML, XML files
- ▶ But: sometimes we need to encode binary data over Character channels
- ▶ **Question:** when does this need arise?
  - ▶ Solution: use base-64
  - ▶ This encodes arbitrary byte sequences using bytes allowed character sequences
  - ▶ The cost is that it requires 1/3 more space
  - ▶ Each 3 bytes of binary require 4 bytes of base 64

## FILE STORAGE

- ▶ First we'll look at some Android specific features of file storage
- ▶ This relates to where files can be opened
- ▶ And what permissions are required
- ▶ Then we'll move on to more general points about file storage
  - ▶ In particular, storing data in files with minimal programming effort

## ANDROID FILE SYSTEM

- ▶ <http://developer.android.com/training/basics/data-storage/files.html>
- ▶ Internal versus External Storage
- ▶ Internal:
  - ▶ By default readable and writable only by this App
  - ▶ Other more liberal modes have been deprecated
  - ▶ Every app can access its own internal storage
  - ▶ No need to request permission in the manifest file
  - ▶ All files deleted when an app is removed from a device

## INTERNAL FILES No.1 – DIRECT FILE ACCESS

File Creation and Appending:

```
FileOutputStream fos =
    openFileOutput("test.txt", Context.MODE_PRIVATE); // create new
FileOutputStream fos =
    openFileOutput("test.txt", Context.MODE_APPEND); // append
```

- ▶ **openFileOutput()** is a method of the Context class
- ▶ Activity is a subclass of Context
- ▶ Opening for reading:
  - ▶ **FileInputStream fis = openFileInput("test.txt");**

13 / 49

## INTERNAL FILES No.2

- ▶ Alternatively, can make calls to get a File object and then open it for reading, writing or appending in standard Java ways:
  - ▶ File file = new File(context.getFilesDir(), filename);
- ▶ Replace the call to context.getFilesDir() with context.getCacheDir() for temporary files

14 / 49

## EXTERNAL STORAGE

- ▶ Memory outside of an App's own area
  - ▶ May even be on an SD-card
  - ▶ Therefore an App cannot guarantee access to it
- ▶ Should handle this gracefully!
- ▶ Potentially readable/writable by the user and by other apps
- ▶ An obvious choice for sharing data

15 / 49

## USING EXTERNAL STORAGE

- ▶ Add the permission to the manifest file:

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

- ▶ Replace with .READ\_EXTERNAL\_STORAGE for read-only
- ▶ Then use file system in standard Java ways

16 / 49

## MORE ON EXTERNAL STORAGE

- Check availability before using:

```
/* 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;
}
```

- Similar method for Read Only test – see docs

17 / 49

## AD HOC FILE FORMATS

- Data is written out in an entirely application specific way
- Conventions are adopted or invented on the fly by the programming team
  - Very flexible
  - Can choose exactly what data to write and how to format it
- Hard work:
  - May need lots of lines of code
  - And careful effort is needed to keep reader and writer in perfect harmony!
- Okay for simple cases, not for complex Apps

18 / 49

## OBJECT SERIALIZERS

- Unlike application specific formats, these read and write a wide variety of Object structures
  - In a domain-independent way
  - Some of them may have readers and writers in a variety of languages: hence can exchange object data between different languages
  - If they do what you want, they:
    - Are easy to use
- Involves minimal programming effort
- We'll look at three examples...

19 / 49

## JAVA'S NATIVE OBJECT SERIALISATION

- If a class that “implements” Serializable
  - Then Objects of that class can automatically be written to and read from Object streams
  - Very easy
  - Fast binary format, low storage space
  - Handles circular references
- But:
  - Can be hard to recover objects if the classes change
  - Restricted to Java
  - Not human readable
  - Binary format cannot directly be sent over text channels

20 / 49

## XML SERIALIZERS E.G. WOX

- ▶ WOX = Web Objects in XML
- ▶ Java version by Lucas (2004) and extended by Jaimez: addition of base-64 for byte arrays and C# readers and writers
- ▶ Handles objects of most classes
- ▶ XML-based, so not as compact as plain text or as binary
- ▶ But given that it's XML, is efficient
- ▶ Important: handles circular references

21 / 49

## JSON = JAVASCRIPT OBJECT NOTATION

- ▶ Simple lightweight text-based standard for reading-and writing objects
- ▶ Efficient and compact
- ▶ Supported by MANY languages and platforms
- ▶ In many cases the best option except for:
- ▶ Binary formats (image, audio, video)
- ▶ Object graphs with circular references
- ▶ Infuriatingly, JSON cannot handle these
- ▶ However, might be worth trying to work around this...

22 / 49

## JSON CONTINUED

- ▶ I recommend the GSON library from Google for using JSON
- ▶ Very easy to use
- ▶ However, due to limitations of GSON format there are some cases it does not handle easily
  - ▶ E.g. When the declared type of a field is an interface type
  - ▶ ( Can customise it to cope with this, but this is extra work
- ▶ (WOX handles those cases easily)

23 / 49

## GSON SAMPLE (PART 1; FROM DOC)

```

public class GsonTest {
    public static void main(String[] args) {

        // (Serialization)
        BagOfPrimitives obj = new BagOfPrimitives();
        Gson gson = new Gson();
        String json = gson.toJson(obj);
        // ==> json is {"value1":1,"value2":"abc"}

        // Note that you can not serialize objects with
        // circular references since that will result in infinite recursion.

        System.out.println(json);
        // (Deserialization)
        BagOfPrimitives obj2 =
            gson.fromJson(json, BagOfPrimitives.class);
    }
}

```

24 / 49

## GSON TEST CLASS: BAG OF PRIMITIVES (BUT WORKS FOR REFERENCE TYPES ALSO)

```

static class BagOfPrimitives {
    private int value1 = 1;
    private String value2 = "abc";
    private transient int value3 = 3;

    public static String test = "BOO";

    BagOfPrimitives() {
        // no-args constructor
    }
}

```

25 / 49

## SQL DBS AND ANDROID

- ▶ If your app needs to store and retrieve data in a flexible way
  - ▶ Consider using a relational DB
  - ▶ Helps ensure data integrity
  - ▶ Well designed relations cannot store self-inconsistent data
- ▶ Standard SQL language
  - ▶ Very powerful for sorting and selecting the data needed
  - ▶ For simple apps using SQL is harder work than simply writing data in JSON format to file system
  - ▶ But worth the effort when appropriate

26 / 49

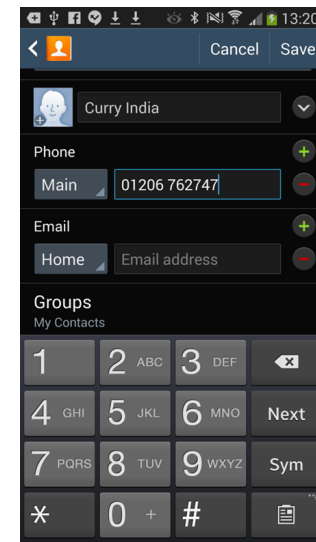
## RELATIONAL MODELLING

- ▶ Relation == table
- ▶ Each column specifies the column name and type
- ▶ Database Schema (loosely speaking)
- ▶ The column names and types for each table
- ▶ Each row is a single record of data with a value for each column
- ▶ Depending on settings, cell entries may be NULL
- ▶ Dilemmas sometimes arise regarding how far to normalise a table

27 / 49

## NORMALISATION

- ▶ How would you model the following Contact DB?
- ▶ Each person is identified by name and primary email address
  - ▶ Each person may have a number of telephones (home, office, mobile etc.)
  - ▶ When designing an App be prepared to compromise:
    - ▶ Perfect normalisation versus realistic usage
    - ▶ Higher normal forms can sometimes be less efficient



28 / 49

## SQLITE

- ▶ Separate open-source project
  - ▶ On Android platform as standard
  - ▶ Hence default choice of relational DB for Android
  - ▶ Other choices of pure Java DB also possible
- ▶ But would add to size of App
  - ▶ Standalone DB
  - ▶ Does not run as a separate server process
  - ▶ Supports most but not all SQL statements
- ▶ Transactions

29 / 49

## TRANSACTIONS

- ▶ Help ensure that DB is always kept in a consistent state
- ▶ Each transaction maps the state of the data from one consistent state to the next
- ▶ ACID properties

30 / 49

## ATOMICITY

- ▶ A transaction may involve a set or sequence of updates
- ▶ Atomicity ensures that either that ALL happen, or NONE of them happen
- ▶ Enclose sequence between begin transaction and end transaction statements
- ▶ Example: imagine the phone battery died during a transaction
- ▶ All the statements executed so far are held in temporary storage
- ▶ And only committed at the end (e.g. by moving a file pointer)

31 / 49

## CONSISTENCY

- ▶ The database is always kept consistent
- ▶ Helped by:
  - ▶ Suitably high normal form
- ▶ Other transactional properties: Atomicity and Isolation

32 / 49



## ISOLATED

- ▶ When concurrent threads are hitting the DB
- ▶ Or data is being processed in App using concurrent threads
- ▶ Must ensure that threads do not interfere
  - ▶ (see example in Threads and Surface Views lecture notes)
- ▶ Isolated / Isolation property guarantees this
- ▶ May be achieved using locking mechanisms such as Semaphores, or Java statements such as synchronized

33 / 49

## DURABLE

- ▶ An obvious property!
- ▶ Once made a transaction should be saved permanently
- ▶ And not be affected by systems crashes or power loss

34 / 49

## PACKAGE: ANDROID.DATABASE.SQLITE

- ▶ Contains the SQLite database management classes that an application can use to manage its own private database
- ▶ Also has the sqlite3 database tool in the tools/ folder
- ▶ Use this tool to browse or run SQL commands on the device. Run by typing sqlite3 in a shell window.

35 / 49

## SQLITE VERSUS JDBC

- ▶ Android has a set of non-JDBC classes to work directly with SQLite
  - ▶ However, JDBC drivers are also available for SQLite
  - ▶ See: <https://github.com/SQLDroid/SQLDroid>
  - ▶ [http://en.wikibooks.org/wiki/Java\\_JDBC\\_using\\_SQLite/Introduction](http://en.wikibooks.org/wiki/Java_JDBC_using_SQLite/Introduction)
- ▶ Hence another possible option would be to use a JDBC driver
- ▶ This offers better standardisation, and could be worth exploring
- ▶ For these notes we're going to stick with the Android API

36 / 49

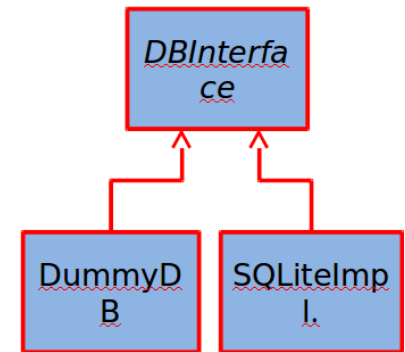
## ACCESSING DBs FROM CODE

- ▶ A number of approaches possible
  - ▶ Can embed SQL strings in App
  - ▶ And make DB calls from wherever needed
- ▶ Or:
  - ▶ Define a special data handling class
  - ▶ All DB access is via handler
  - ▶ Main App code only sees objects, never SQL strings
- ▶ Or:
  - ▶ Can use an automated / semi-automated tool such as Spring / Hibernate
- ▶ Discussion question: which way is best?

37 / 49

## DB INTERFACE / HELPER APPROACH

- ▶ Define all DB access methods in an interface
- ▶ Then provide one or more implementations of this as required e.g.:
- ▶ SQLite Implementation: the real thing
  - ▶ DummyDB: implement all the methods but don't actually save the data persistently
  - ▶ DummyDB can be very useful for testing the App



38 / 49

## EXERCISE

- ▶ Write a DB interface class called ScoreDBInterface to support the storage and retrieval of all scores for a game
- ▶ Interface should allow storage of each score as it occurs
- ▶ And retrieval of high score, and list of top N scores
- ▶ Each score record consists of:
  - ▶ Time Stamp of type long
  - ▶ Player Name of type String
  - ▶ Score achieved of type int
- ▶ Write the method signatures and also any convenience classes that you need

39 / 49

## ANDROID APPROACH

- ▶ Similar to DBInterface approach
- ▶ Defines data contract class
  - ▶ This includes an inner class for each table defining its column names
  - ▶ Methods to create tables and drop them
  - ▶ Methods to execute queries
- ▶ See Android Developer example
  - ▶ <http://developer.android.com/training/basics/data-storage/databases.html>
  - ▶ Also CE881 Lab code, outlined below

40 / 49

## SQLITE IN ANDROID

- ▶ Construct SQL strings
- ▶ Execute SQL strings to perform actions
  - ▶ Also use specific helper methods such as query()
- ▶ These take arguments to build a SELECT clause with
- ▶ Or use.rawQuery() – pass the SQL string
- ▶ Both query() and.rawQuery() return a Cursor object
- ▶ This is used to iterate over the resulting table of rows

41 / 49

## SCORE DB EXAMPLE

- ▶ Adapted from FeedReader example:
- ▶ <http://developer.android.com/training/basics/data-storage/databases.html>
- ▶ ScoreDBContract
  - ▶ Class used to define table names
  - ▶ Has inner classes for each table
  - ▶ In this case, just one table called entry (for entries in the score table)
  - ▶ All columns defined in abstract class **ScoreEntry**
- ▶ ScoreHelper class
  - ▶ Manages access to the DB
  - ▶ Declares types for each table column
  - ▶ Methods for creating and dropping tables
  - ▶ May also implement **ScoreDBInterface**

42 / 49

## SNIPPETS FROM SCOREHELPER

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + TABLE_NAME + " (" +
        ScoreDBContract.ScoreEntry._ID +
        " INTEGER PRIMARY KEY," +
        ScoreDBContract.ScoreEntry.COLUMN_NAME_ENTRY_ID +
        TEXT_TYPE + COMMA_SEP +
        ScoreDBContract.ScoreEntry.COLUMN_NAME_PERSON +
        TEXT_TYPE + COMMA_SEP +
        ScoreDBContract.ScoreEntry.COLUMN_NAME_SCORE +
        "INTEGER" + " ) ";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + TABLE_NAME;

public void onCreate(SQLiteDatabase db) {
    db.execSQL(SQL_CREATE_ENTRIES);
}
```

43 / 49

## ADDING AN ENTRY...

```
SQLiteDatabase db =
    scoreHelper.getWritableDatabase();
scoreHelper.addEntry(db, "Simon", "" +
    10 * (1 + random.nextInt(100)));
Log.i(TAG, "Populated Table");
db.close();
```

44 / 49

## ADDENTRY METHOD

```
public void addEntry(SQLiteDatabase db, String person, String score) {
    // Create a new map of values, where column names are the keys
    ContentValues values = new ContentValues();
    values.put(ScoreDBContract.ScoreEntry.COLUMN_NAME_ENTRY_ID, id++);
    values.put(ScoreDBContract.ScoreEntry.COLUMN_NAME_PERSON, person);
    values.put(ScoreDBContract.ScoreEntry.COLUMN_NAME_SCORE, score);

    // Insert the new row, returning the primary key value of the new row
    long newRowId;
    newRowId = db.insert(
        ScoreDBContract.ScoreEntry.TABLE_NAME,
        ScoreDBContract.ScoreEntry.COLUMN_NAME_NULLABLE,
        values);
    Log.i(SQLiteActivity.TAG, "Inserted row: " + newRowId);
}
```

45 / 49

## USING A CURSOR WITH A QUERY SELECTING ALL SCORES

```
Cursor cursor = db.rawQuery("Select * from " +
    ScoreDBContract.ScoreEntry.TABLE_NAME,
    new String[]{});
cursor.moveToFirst();
// need to find index of each column before retrieving it
int scoreIndex =
    cursor.getColumnIndex(ScoreDBContract.ScoreEntry.COLUMN_NAME_SCORE);
while (!cursor.isAfterLast()) {
    int score = cursor.getInt(scoreIndex);
    boolean flag = cursor.moveToNext();
    Log.i(TAG, flag + " : " + score);
}
```

46 / 49

## MORE DETAILS...

- ▶ See SQLite lab exercise
  - ▶ Including .zip with all the details
  - ▶ The example opens and closes a DB Connection each time it is needed
  - ▶ This is perhaps not the most efficient way
  - ▶ But it saves thinking through lifecycle methods
- ▶ More efficient way:
  - ▶ open connection when activity is created
  - ▶ Close connection when activity is destroyed

47 / 49

## SUMMARY (1)

- ▶ Data serialization is an important topic
- ▶ These notes discussed storage and retrieval on file systems
- ▶ But much of this same applies for sending over a network
- ▶ Choose carefully: when given a free choice JSON is a good default

48 / 49

## SUMMARY (2)

- ▶ Android ships with SQLite database
- ▶ Natural choice for storing data that must be flexibly queried
- ▶ And have guaranteed integrity
- ▶ However, API is rather low level
- ▶ Best approach is to embed all DB access code in a separate helper class (or classes)
- ▶ My favourite: all access goes through a DB interface class
- ▶ Can then test rest of app using a dummy implementation
- ▶ Work through this week's lab exercise
- ▶ Some slides by Simon Lucas