Mobile Application Development (COMP2008)

Lecture 4: Local Data

Updated: 6th September, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

First compare quality. Then compare cost.

Morrow Designs' 10 megabyte hard disk system: $3,695.

MORE MEMORY. LESS MONEY.

Compare Morrow Designs' DISCUS™ M26™ hard disk systems to any system available for S-100 or Cromemco machines. First, compare features. Then, compare cost per megabyte. The M26 works out to under $200 a megabyte. And the M10 is about half the cost of competing systems.

COMPLETE SUBSYSTEMS.

Both the M10 (8"), and the M26 (14"), are delivered complete with disk controller, cables, fan, power supply, cabinet and CP/M® operating system. It's your choice: 10 Mb 8" at $3,695 or 26 Mb 14" at $4,995. That's single unit. Quantity prices are available.

(http://www.portmacquariewebdesigns.com.au/blog/technology/a-10-megabyte-hard-drive-for-only-3495-classic-computer-ads/attachment/vintage-computer-ad6/.)

Outline

Databases

Defining the Schema

Creation and Upgrading

Inserting, Updating and Deleting

Querying

Local Storage

- Most apps need permanent local storage.
  - They need to remember things!
- Android gives each app a private directory to store things in.
  - In principle, we can do whatever we like here.
  - *However*. . .
- In practice, we generally want to store *structured* data.
- And this is what a database is for.
  - How would you store all the data from worksheet 2?
  - Sure, you could invent your own text file format.
  - But databases can be a more flexible and maintainable solution.

Android File IO

- First, all Activity classes inherit some basic IO methods:
- openFileInput(String name), returns a FileInputStream.
  - You can wrap this in an InputStreamReader and a BufferedReader (for instance) to read a text file:

```
try(BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                        openFileInput("file.txt")))
{
    ... // Read file!
}       // (See JDK API docs for BufferedReader.)
```

- openFileOutput(String name, int mode):
  - "mode" should generally be MODE_PRIVATE or MODE_APPEND.
  - Returns FileOutputStream.
  - Wrap this in a PrintWriter (for instance) to write a text file.

Database Systems

- Many/most of you will have seen databases before, in Database Systems (or elsewhere).
- However, MAD does not have a pre-requisite on Database Systems.
- So we have to tread carefully.
- We're only going to do the following:
  - Create a database table;
  - Insert data;
  - Update existing data;
  - Delete data;
  - Retrieve (query) data from a single table.

Database Tables

- Just to make sure you're on the same page. . .
- An database[1] is made of tables.
- Each table has a name, and several rows and columns.
- The rows are records.
  - Applications call on the database to add, update, remove and retrieve rows as needed.
  - Often each row represents the same thing as a model object.
- The columns are fields.
  - Each represents a different *aspect* of the data.
  - Each has a name.
  - Each has potentially a different data type.
    - (Though whether this is actually enforced is another question.)

[1] Specifically a *relational* / SQL-based database.

SQLite

- SQLite is a very popular and very simple database system.
  - Its website claims (quite plausibly) that it is the most widely used database engine in the world.
- *Not* server-based.
  - You don't "connect" to an SQLite database over a network.
  - Rather, it's a library that stores everything in a local file.
- *Not* a sophisticated, high-performance database!
  - Don't use it to store large volumes of critical data.
  - Don't use it as the "back end" to an important web application.
- Designed for simplicity, on a private, single-user system.
- Embedded into Android, and a standard way of storing app data.

[1] https://www.sqlite.org/index.html

What's Involved?

- We (should) first create a "schema" class just to define the *names* of things as constants.
  - This is for maintainability purposes, and to help avoid bugs due to spelling mistakes.
- We must define how to create and upgrade a database.
- We place any insert, update, remove, and query operations in our main model class.
- We must define a "cursor" to (when necessary) extract data and build objects out of it.

The Schema Class

- There are any number of possible ways to do this.
- I'm taking the convention outlined in the Big Nerd Ranch Guide.

```
public class PetStoreSchema
{
    public static class PetTable
    {                                             // Table
        public static final String NAME = "pets"; // name
        public static class Cols
        {                                  // Column names
            public static final String ID = "pet_id";
            public static final String TYPE = "type";
            public static final String PRICE = "price";
            public static final String DESCR = "descr";
        }
    }
}
```

The Schema Class – Usage

- The schema class exists purely for its constants.
- We now write “PetStoreSchema.PetTable.NAME”, for the table name.
  - We could just write ”pets”, of course.
  - But the compiler doesn’t do any checking on literal strings.
  - If we accidentally write just ”pet”, the app could crash, or otherwise misbehave.
  - This can’t happen if we use a constant – the compiler will complain immediately.
- We can make it simpler by importing the nested class directly:

```
import com.example.myapp.PetStoreSchema.PetTable;
```

Now we can just write “PetTable.NAME”.

Database Creation and Upgrading

- Your app's database first needs to be created.
  - This is done *at runtime* by your application.
  - However, it is only done once! Don't re-create the database if one already exists, or you'll lose all the data.
- We also need to plan ahead. Your database must have a version number, starting at 1.
- Why?
- Because, your database structure will need to change in the future.
  - Not just the rows, but the columns and tables.
  - This is practically inevitable – it's the nature of software maintenance.
- Upgrading a database requires migrating the data itself.
  - Upgrading code is easy – you just replace it.
  - But you have to preserve the data in a database – if you throw it out, your users will come looking for you!

SQLiteOpenHelper

- An abstract class from the Android API.
- This is a starting point for creating/upgrading your database:

```
public class PetStoreDbHelper extends SQLiteOpenHelper
{
    private static final int VERSION = 1;
    private static final String DATABASE_NAME = ”pets.db”;

    public PetStoreDbHelper(Context context)
    {
        super(context, DATABASE_NAME, null, VERSION);
    }

    @Override public void onCreate(SQLiteDatabase db) {...}
    @Override public void onUpgrade(SQLiteDatabase db,
        int v1, int v2) {...}
}
```

Creation

```
@Override                              // Inside PetStoreDbHelper
public void onCreate(SQLiteDatabase db)
{
    db.execSQL("create table " + PetTable.NAME + "(" +
               " _id integer primary key autoincrement, " +
               PetTable.Cols.ID + ", " +
               PetTable.Cols.TYPE + ", " +
               PetTable.Cols.PRICE + ", " +
               PetTable.Cols.DESCR + ")");
}
```

- This is ultimately the string we're passing to `db.execSQL()`:

```
create table pets(
    _id integer primary key autoincrement,
    pet_id, type, price, descr)
```

- SQLite allows but doesn't require datatype specifications.
- "_id" is a separate, auto-generated ID column.

Upgrading

- Upgrading only becomes necessary once your app starts getting used "for real", to store data that you cannot afford to delete.
  - During development, you're *not* doing this (hopefully).
  - If you need to change the database, you *can* just un-install the app, and let Android Studio re-install it.
- We're not going to tackle the problem of upgrading a database schema here.
- It's going to be highly situation-dependent, and possibly very complex.

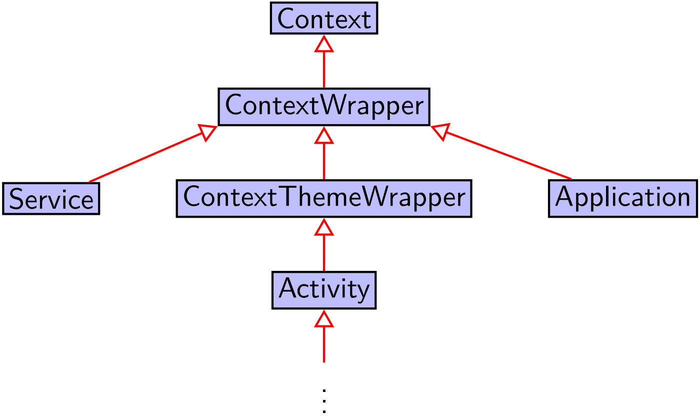
The Model Class

- Make one class (*not* an activity, fragment, etc.) responsible for keeping track of your data.
  - There will be various classes *representing* the data, as you've already been using.
  - But there should be one main one.
- This is where the database interaction will go.

```
public class PetStore
{
    private SQLiteDatabase db;
    public PetStore(Context context)
    {
        this.db = new PetStoreDbHelper(
            context.getApplicationContext()
            ).getWritableDatabase();
    }
    ...
}
```

The Context (a slight diversion)

- What is a "`Context`" object?
- "Allows access to application-specific resources and classes, . . . operations such as launching activities, broadcasting and receiving intents, etc." (API docs)



Inserting Data

- We insert data one row at a time, using ContentValues.
- SQLiteDatabase.insert() sends an INSERT statement to the database.

```
public void addPet(Pet pet)                // Inside PetStore
{
    ContentValues cv = new ContentValues();
    cv.put(PetTable.Cols.ID, pet.getId());
    cv.put(PetTable.Cols.TYPE, pet.getType());
    cv.put(PetTable.Cols.PRICE, pet.getPrice());
    cv.put(PetTable.Cols.DESCR, pet.getDescription());
    db.insert(PetTable.NAME, null, cv);
}
```

- The middle (null) parameter is a hack to handle inserting an empty row.
  - We're not actually using it here, as we don't need to.
  - See the API documentation for more information.

Updating Data

- SQLiteDatabase.update() sends an UPDATE statement.

```
public void updatePet(Pet pet)             // Inside PetStore
{
    ContentValues cv = ...; // As before
    ...
    String[] whereValue = { String.valueOf(pet.getId()) };
    db.update(PetTable.NAME, cv,
        PetTable.Cols.ID + ” = ?”, whereValue);
}
```

- The last two arguments say which record to update.
- update() constructs an SQL “WHERE” clause.
- Each “?” is replaced by the next index in the string array.
  - In most case, we’ll only need to specify one column value.
- String.valueOf() is just converting the ID (integer) to a string. Different situations may require different conversions.

Deleting Data

- SQLiteDatabase.delete() works like update(), except you don’t specify any updated values:

```
public void removePet(Pet pet)             // Inside PetStore
{
    String[] whereValue = { String.valueOf(pet.getId()) };
    db.delete(PetTable.NAME,
        PetTable.Cols.ID + ” = ?”, whereValue);
}
```

Retrieving Data

- Getting data from a database uses the “SELECT” statement.
  - Generally, SELECT queries can be enormously complex (as you may know from Database Systems, etc.).
  - In MAD, we’re only going to use *very, very* simple ones!
- The returned data has its own rows and columns.
- We use a “cursor” to iterate over it.
  - A cursor is (broadly) the same idea as an iterator.
  - At any given time, the cursor is “at” a particular row (of the returned data).
    - You tell it to move between rows.
  - For any given row, you can access the column values.
    - But you first have to obtain the “column index” for a given column name.
  - It’s best to create our own cursor subclass, to hide the low-level details.

Defining a Cursor Subclass

```
public class PetCursor extends CursorWrapper
{
    public PetCursor(Cursor cursor) { super(cursor); }

    public Pet getPet()
    {
        int id = getInt(getColumnIndex(PetTable.Cols.ID));
        String type = getString(
            getColumnIndex(PetTable.Cols.TYPE));
        double price = getDouble(
            getColumnIndex(PetTable.Cols.PRICE));
        String descr = getString(
            getColumnIndex(PetTable.Cols.DESCR));
        return new Pet(id, type, price, descr);
    }
}
```

Executing the Query

- SQLiteDatabase.query() performs a SELECT query.
- It takes a parameter for each part of the standard SELECT.
- To retrieve an entire database table, we can set most of them to null:

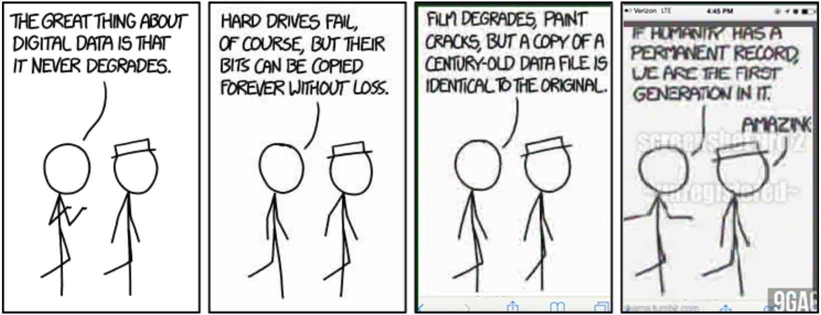
```
PetCursor cursor = new PetCursor(
    db.query(PetTable.NAME, // FROM our table
             null, // SELECT all columns
             null, // WHERE clause (null = all rows)
             null, // WHERE arguments
             null, // GROUP BY clause
             null, // HAVING clause
             null) // ORDER BY clause
);
```

Iterating Over Query Results

```
List<Pet> pets = new ArrayList<>();
PetCursor cursor = ...; // Execute query
try
{
    cursor.moveToFirst();
    while(!cursor.isAfterLast())
    {
        pets.add(cursor.getPet()); // From previously
        cursor.moveToNext();
    }
}
finally
{
    cursor.close(); // This is needed, or your app will
}                   // "leak" certain resources.
```

In-Database or In-Memory?

- Once everything is in a database, technically you don't *need* it in memory too.
- Instead of having `List<Pet>`, you could just set/get data directly to/from the database.
- But database operations are inherently slower than in-memory operations.
  - Scrolling through a `RecyclerView` list may not work as smoothly, if the adapter is getting things straight from the database.
- Typically you *want* to keep (at least some) data in memory
- But this does mean you must know when to load/re-load it.


THE GREAT THING ABOUT DIGITAL DATA IS THAT IT NEVER DEGRADES.
HARD DRIVES FAIL, OF COURSE, BUT THEIR BITS CAN BE COPIED FOREVER WITHOUT LOSS.
FILM DEGRADES, PAINT CRACKS, BUT A COPY OF A CENTURY-OLD DATA FILE IS IDENTICAL TO THE ORIGINAL.
Verizon LTE
4:45 PM
IF HUMANITY HAS A PERMANENT RECORD, WE ARE THE FIRST GENERATION IN IT.
AMAZING
9GAG

(https://xkcd.com/1683/)