IT332: Mobile Application Development

Lecture # 18 : SQLite Databases

Muhammad Imran

Outline

- SQLite Databases
- Defining a Schema
- Building Initial Database
- Creating a Table
- Debugging database issues
- Writing to Database
- Inserting and updating rows
- Reading from the Database
- Using a CursorWrapper
- Finalizing

Device’s Sandbox

- Almost every application needs a place to **save data** for the **long term**, longer than **savedInstanceState**
- Android provides a **local filesystem** on the phone flash memory storage.
- Each application on Android device has a **directory** in the **device’s sandbox**.
- Keeping files in the sandbox **protects** from being accessed by other applications
- Each application’s sandbox directory is a child of the device’s **/data/data** directory named after the application package.

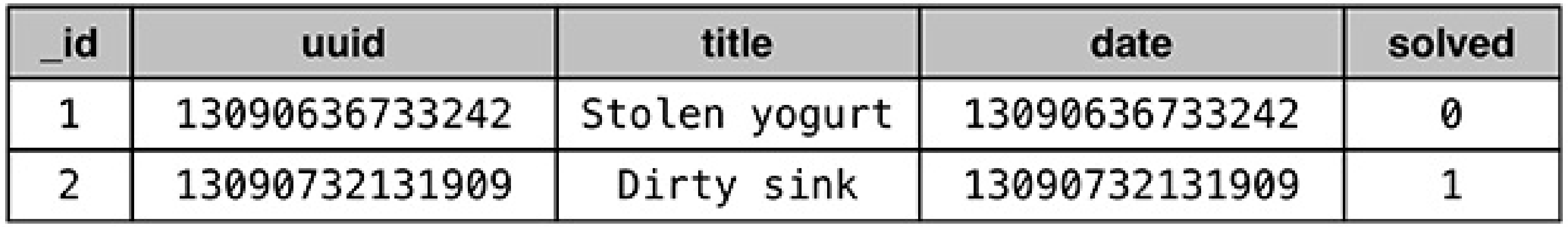
- For CrimeReporting App, the **full path** to the sandbox directory is **/data/data/com.nomadlearner.crimereporting**

SQLite Databases

- SQLite is an open source **relational database**, like MySQL
- SQLite stores its data in simple **files**, which we can read and write using the **SQLite library**
- Android includes this SQLite library in its **standard library**, along with some additional Java helper classes.

Defining a Schema

- Before we create a database, we have to decide what will be in that database.
- CrimeReporting Application stores a single list of crimes, so we will define one table named crimes



_id	uuid	title	date	solved
1	13090636733242	Stolen yogurt	13090636733242	0
2	13090732131909	Dirty sink	13090732131909	1

- There are even complex tools called object-relational mappers (ORMs) that let us use our model objects (like Crime)

Defining a Schema

- We will creating a class **CrimeDbSchema** to put our schema in, but in the Create New Class dialog, we name it database.CrimeDbSchema.
- This will put the CrimeDbSchema.java file in its own database package, which can be used to organize all database-related code.
- Inside CrimeDbSchema, an inner class **CrimeTable** will describe the table.

```
public class CrimeDbSchema {
    public static final class CrimeTable {
        public static final String NAME = "crimes";
    }
}
```

- The CrimeTable class defines the String constants needed to describe the moving pieces of your table definition.
- The first piece of that definition is the name of the table, **CrimeTable.NAME**.

Defining your table columns (CrimeDbSchema.java)

```
public class CrimeDbSchema {
    public static final class CrimeTable {
        public static final String NAME = "crimes";

        public static final class Cols {
            public static final String UUID = "uuid";
            public static final String TITLE = "title";
            public static final String DATE = "date";
            public static final String SOLVED = "solved";
        }
    }
}
```

- With that, we refer to the column names like “title” in a Java-safe way:
  - **CrimeTable.Cols.TITLE** makes it much safer to update the name of column or add additional data to the table.

Building Initial Database

- After defining the schema, we can create the database itself.
- Android provides some low-level methods on Context to open a database file into an instance of SQLiteDatabase:
  - **openOrCreateDatabase(…)** and **databaseList()**.
- We need to follow a few basic steps:
  - Check to see whether the database **already exists**.
  - If it **does not,** create it and create the tables and initial data it needs.
  - If it **does**, open it up and see what version of your CrimeDbSchema it has. (You may want to add or remove things in future versions of CrimeReporting App)
- If it is an old version, **upgrade** it to a newer version.

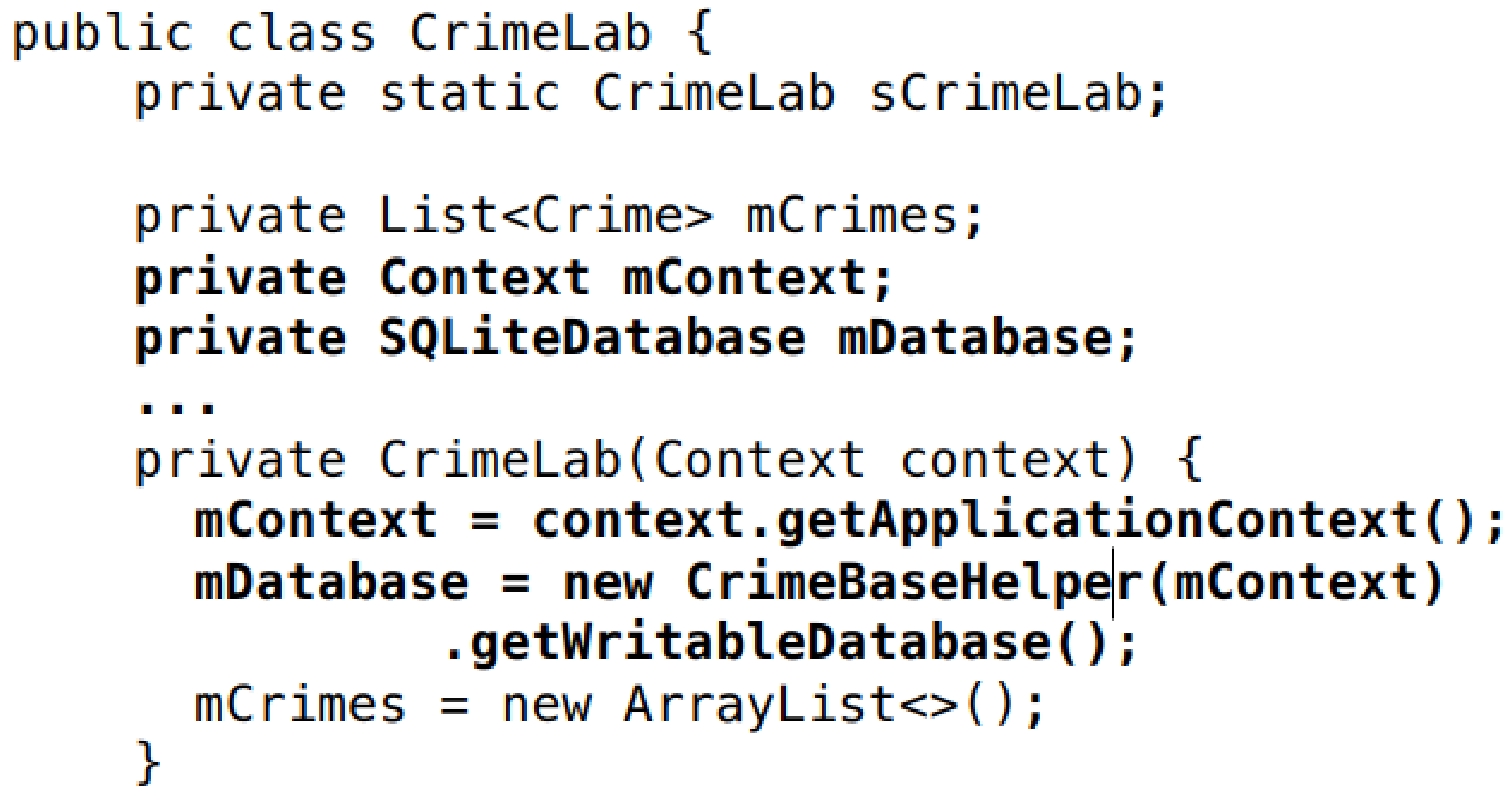
Building Initial Database

- Android provides the **SQLiteOpenHelper** class to handle SQLLite DB.
- We can create a class called **CrimeBaseHelper** in the database package.

```
public class CrimeBaseHelper extends SQLiteOpenHelper {
    private static final int VERSION = 1;
    private static final String DATABASE_NAME = "crimeBase.db";

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


    @Override
    public void onCreate(SQLiteDatabase db) {

    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }
}
```

Opening a SQLiteDatabase (CrimeLab.java)

- We can use our CrimeBaseHelper class inside of CrimeLab to create our crime database.



```java
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    private List<Crime> mCrimes;
    private Context mContext;
    private SQLiteDatabase mDatabase;
    ...
    private CrimeLab(Context context) {
      mContext = context.getApplicationContext();
      mDatabase = new CrimeBaseHelper(mContext)
              .getWritableDatabase();
      mCrimes = new ArrayList<>();
    }
```

Opening a SQLiteDatabase (CrimeLab.java)

- The **getWritableDatabase()** will do the following:
  - **Open up** /data/data/com.nomadlearner.crimereporting/databases/crimeBase.db, or **create a new** database file if it does not already exist.
  - If this is the first time the database has been created, call onCreate(SQLiteDatabase), then **save out** the **latest version number**.
  - If this is **not the first time**, check the version number in the database. If the version number in CrimeBaseHelper is higher, call **onUpgrade(SQLiteDatabase, int, int).**

- We put our code to **create the initial database** in **onCreate(SQLiteDatabase)**,
- We put our code to **handle any upgrades** in **onUpgrade(SQLiteDatabase, int, int)**

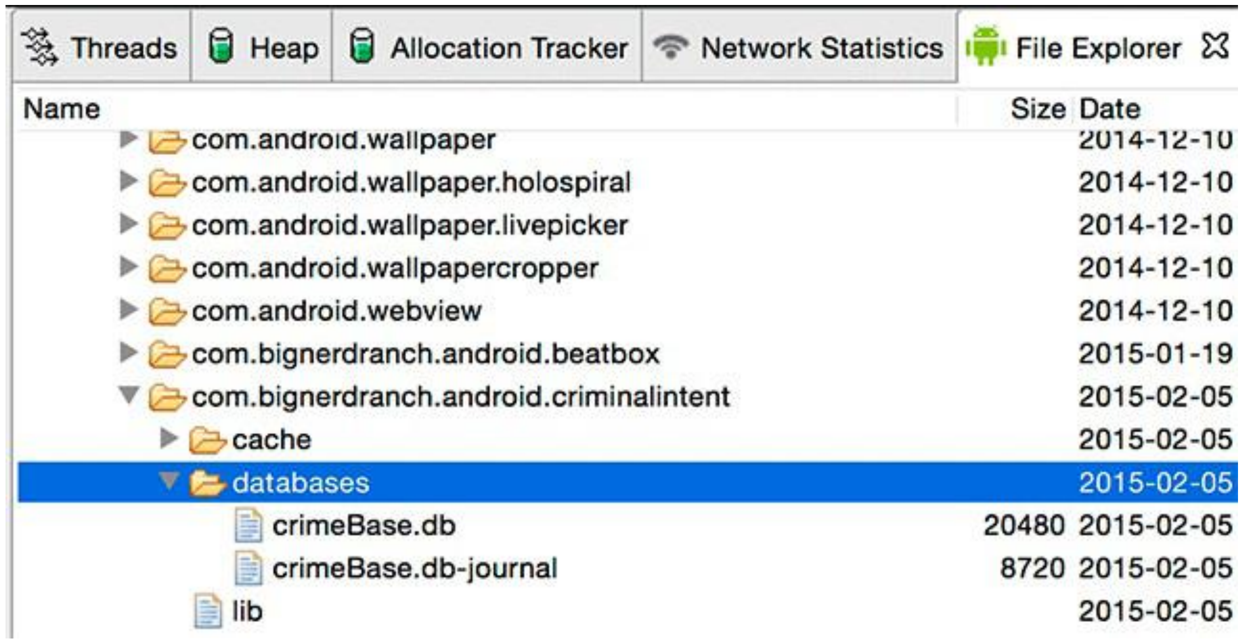
Creating crime table (CrimeBaseHelper.java)

- For now, CrimeReporting app will only have **one version**, so we can **ignore onUpgrade(…).**
- We only need to **create** the database tables in **onCreate(SQLiteDatabase).**
- To do that, we will refer to the CrimeTable **inner class** of **CrimeDbSchema**.

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("create table " + CrimeTable.NAME + "(" +
            " _id integer primary key autoincrement, " +
            CrimeTable.Cols.UUID + ", " +
            CrimeTable.Cols.TITLE + ", " +
            CrimeTable.Cols.DATE + ", " +
            CrimeTable.Cols.SOLVED +
            ")"
    );
}
```

Creating crime table (CrimeBaseHelper.java)

- In SQLite, we **do not** have to specify the type of a column at creation time.
- Now, when we will **run** our application and our database will be **created**.
- On an **emulator** or a **rooted device**, **we can look** at the **DB file** directly. (On an unrooted device, it is saved in private storage, which is secret.)



```
Name                                              Size  Date
  ▶ com.android.wallpaper                               2014-12-10
  ▶ com.android.wallpaper.holospiral                    2014-12-10
  ▶ com.android.wallpaper.livepicker                    2014-12-10
  ▶ com.android.wallpapercropper                        2014-12-10
  ▶ com.android.webview                                 2014-12-10
  ▶ com.bignerdranch.android.beatbox                    2015-01-19
  ▼ com.bignerdranch.android.criminalintent             2015-02-05
    ▶ cache                                             2015-02-05
    ▼ databases                                         2015-02-05
        crimeBase.db                              20480 2015-02-05
        crimeBase.db-journal                       8720 2015-02-05
      lib                                               2015-02-05
```

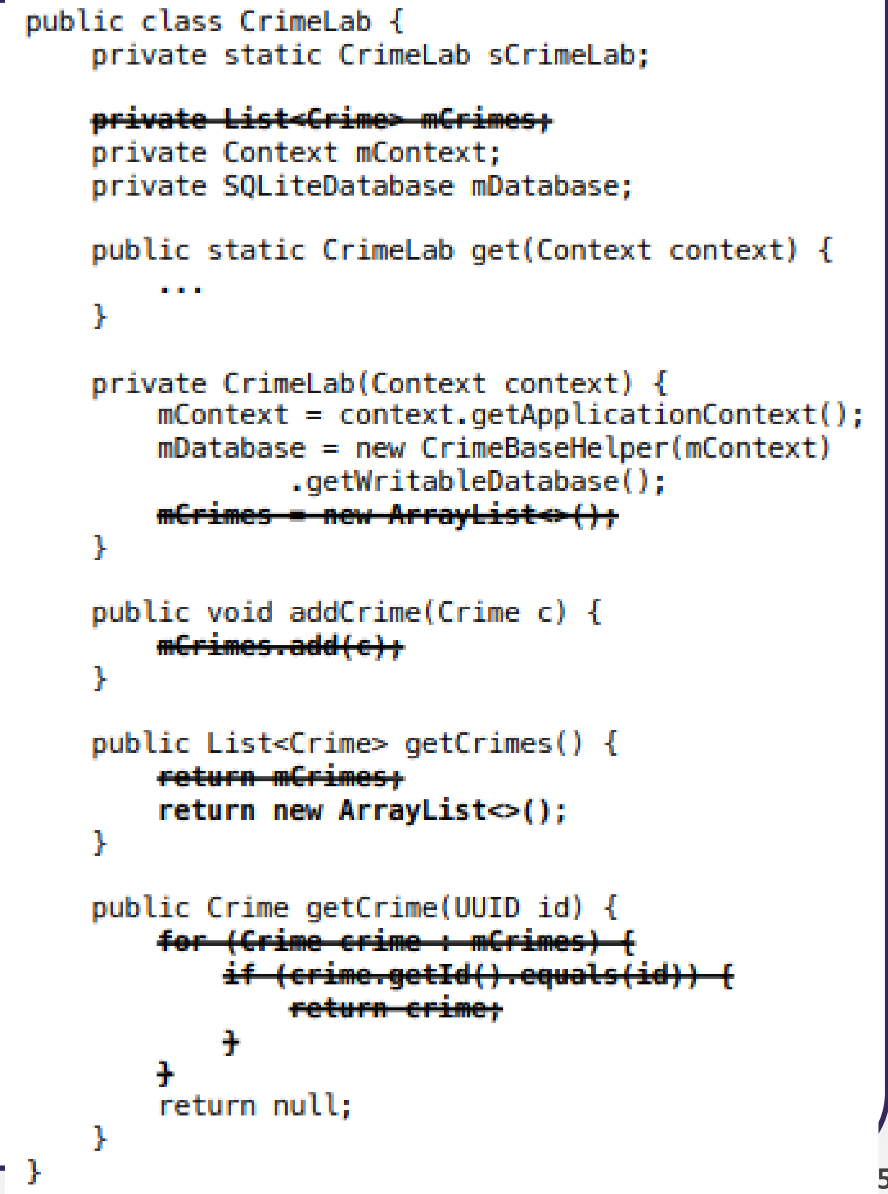
Debugging database Issues

- If we need to change the DB schema, the “right” way to do this is to write code in **SQLiteOpenHelper** to bump the **version number**, and then update the tables inside **onUpgrade(…).**
- But the “right” way involves a fair amount of ridiculous code just to get version 1 or 2 of the database right.
- In practice, the best thing to do is destroy the database and start over, so that **SQLiteOpenHelper.onCreate(…)** is called again.
- The easiest way to destroy database is to delete the app off your device.

- **Remember this trick if you run into any issues with your database tables**

Gutting CrimeLab

- To update our already existing CrimeLab class to **use the DB** instead of an ArrayList, we will modify its code
- We can start by stripping out all the code related to **mCrimes** in **CrimeLab**.



```
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    ~~private List<Crime> mCrimes;~~
    private Context mContext;
    private SQLiteDatabase mDatabase;

    public static CrimeLab get(Context context) {
        ...
    }

    private CrimeLab(Context context) {
        mContext = context.getApplicationContext();
        mDatabase = new CrimeBaseHelper(mContext)
                .getWritableDatabase();
        ~~mCrimes = new ArrayList<>();~~
    }

    public void addCrime(Crime c) {
        ~~mCrimes.add(c);~~
    }

    public List<Crime> getCrimes() {
        ~~return mCrimes;~~
        return new ArrayList<>();
    }

    public Crime getCrime(UUID id) {
        ~~for (Crime crime : mCrimes) {~~
            ~~if (crime.getId().equals(id)) {~~
                ~~return crime;~~
            ~~}~~
        ~~}~~
        return null;
    }
}
```

Writing to the Database

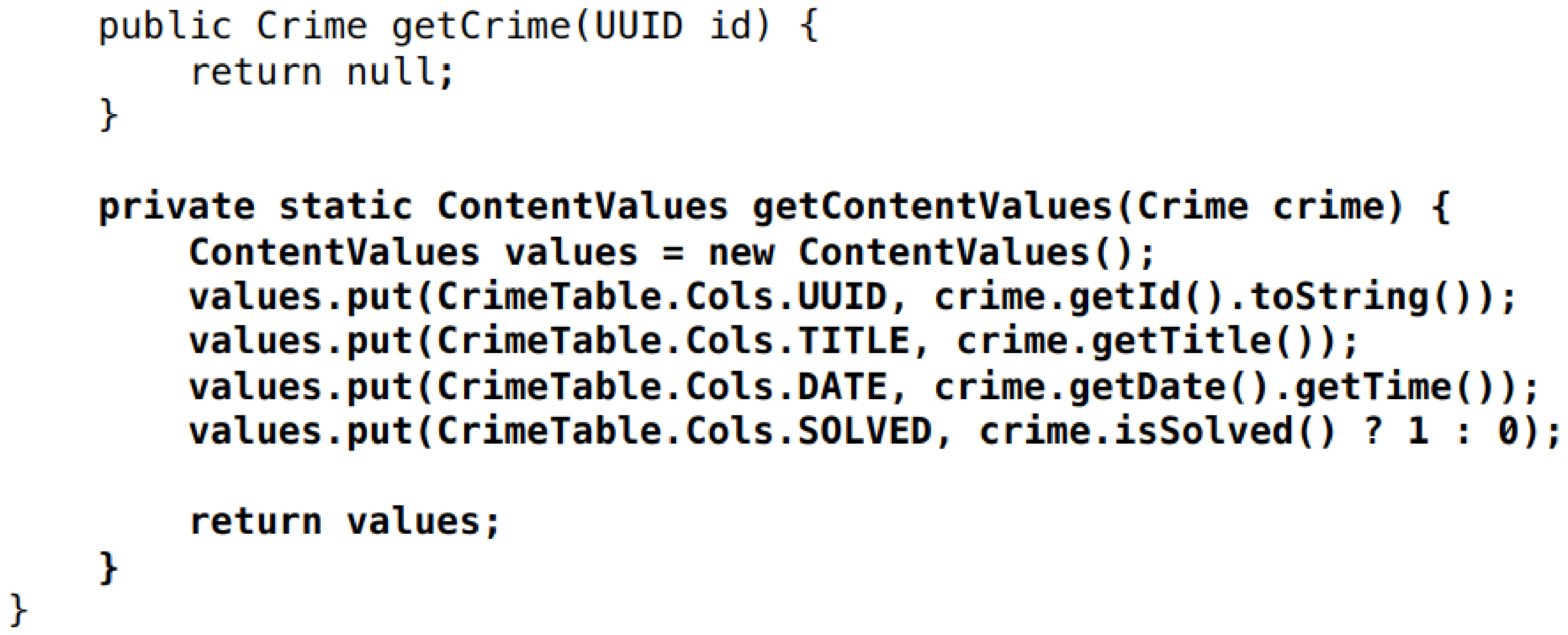
- The first step in using SQLiteDatabase is to **write data** to it.
- We can **insert** new rows into the crime table as well as **update** rows that are already there when Crimes are changed.

Using ContentValues

- Writes and updates to databases are done with the assistance of a class called **ContentValues**.
- ContentValues is a **key-value store** class, like Java's HashMap or the Bundles we have been using so far.
- However, unlike HashMap or Bundle, it is specifically designed to store the kinds of data SQLite can hold.
- For the keys, we use our column names.
- These are not arbitrary names; they specify the columns that we want to insert or update.
- If they are misspelled or typo'd compared to what is in the database, the insert or update will fail.

Creating a ContentValues (CrimeLab.java)

- We will be creating ContentValues instances from Crimes a few times in CrimeLab.
- We will Add a **private method** to take care of shuttling a Crime into a ContentValues.

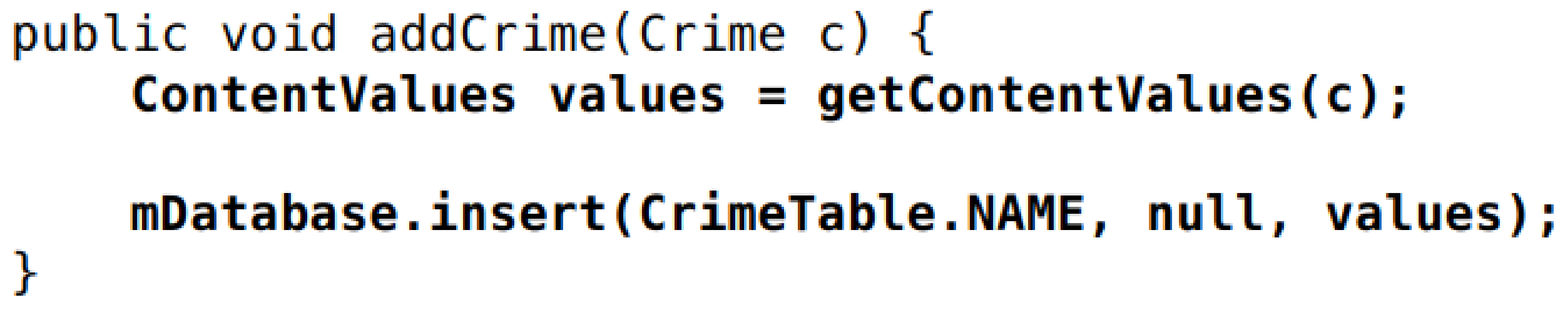


```
    public Crime getCrime(UUID id) {
        return null;
    }

    private static ContentValues getContentValues(Crime crime) {
        ContentValues values = new ContentValues();
        values.put(CrimeTable.Cols.UUID, crime.getId().toString());
        values.put(CrimeTable.Cols.TITLE, crime.getTitle());
        values.put(CrimeTable.Cols.DATE, crime.getDate().getTime());
        values.put(CrimeTable.Cols.SOLVED, crime.isSolved() ? 1 : 0);

        return values;
    }
}
```

- Every column is specified here **except** for **_id**, which is **automatically created** for you as a unique row ID.

Inserting a row (CrimeLab.java)

- We will write **addCrime(Crime)** with a new implementation.



```
public void addCrime(Crime c) {
    ContentValues values = getContentValues(c);

    mDatabase.insert(CrimeTable.NAME, null, values);
}
```

- The **insert(String, String, ContentValues)** method has two important arguments and one that is rarely used.
- The **first argument** is the **table** you want to insert into – here, CrimeTable.NAME.
- The **last argument** is the **data** you want to put in.
- The **second argument** is called **nullColumnHack**.

nullColumnHack

- For example, if we decided to call **insert(…)** with an empty ContentValues, SQLite does not allow this, so the insert(…) call would **fail**.

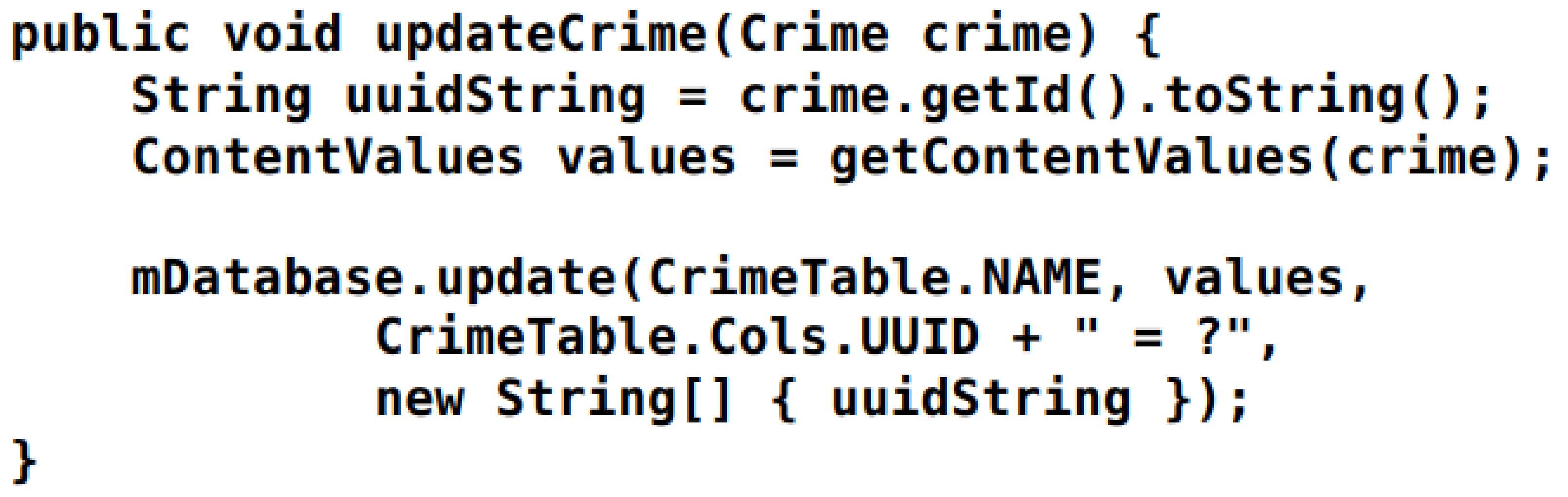
- If we passed in a value of **uuid** for nullColumnHack, though, it would ignore that empty ContentValues.
- Instead, it would pass in a ContentValues with uuid set to null. This would allow your insert(…) to succeed and create a new row.

nullColumnHack

- For example, if we decided to call **insert(…)** with an empty ContentValues, SQLite does not allow this, so the insert(…) call would **fail**.

- If we passed in a value of **uuid** for nullColumnHack, though, it would ignore that empty ContentValues.
- Instead, it would pass in a ContentValues with uuid set to null. This would allow your insert(…) to succeed and create a new row.

Updating a Crime (CrimeLab.java)



```java
public void updateCrime(Crime crime) {
    String uuidString = crime.getId().toString();
    ContentValues values = getContentValues(crime);

    mDatabase.update(CrimeTable.NAME, values,
            CrimeTable.Cols.UUID + " = ?",
            new String[] { uuidString });
}
```

- In **update(String, ContentValues, String, String[])** method we pass in the **table name** and the **ContentValues** for each row we update.
- We have to specify which rows get updated by building a where clause (the **third argument**) and then specifying values for the arguments in the where clause (the final String[] array).

Pushing updates (CrimeFragment.java)

- Crime instances get modified in CrimeFragment and will need to be written out when CrimeFragment is done.
- We will add an **override** to **CrimeFragment.onPause()** that **updates** CrimeLab's copy of the Crime.

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
    mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);
}

@Override
public void onPause() {
    super.onPause();

    CrimeLab.get(getActivity())
            .updateCrime(mCrime);
}
```

Reading from the Database

- Reading in data from SQLite is done using the **query(…)** method.
- SQLiteDatabase.query(…) has different overloads
- The **table** argument is the **table to query**.
- The **columns** argument names **which columns** you want values for and **what order** you want to receive them in.
- And then **where** and whereArgs specify which rows get updated

```
public Cursor query(
      String table,
      String[] columns,
      String where,
      String[] whereArgs,
      String groupBy,
      String having,
      String orderBy,
      String limit)
```

Querying for Crimes (CrimeLab.java)

```
private Cursor queryCrimes(String whereClause, String[] whereArgs) {
    Cursor cursor = mDatabase.query(
            CrimeTable.NAME,
            null, // columns - null selects all columns
            whereClause,
            whereArgs,
            null, // groupBy
            null, // having
            null  // orderBy
    );

    return cursor;
}
```

Using a CursorWrapper

- A **Cursor** give us raw column values.
- Pulling data out of a Cursor looks like this:

```
String uuidString = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.UUID));
String title = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.TITLE));
long date = cursor.getLong(
    cursor.getColumnIndex(CrimeTable.Cols.DATE));
int isSolved = cursor.getInt(
    cursor.getColumnIndex(CrimeTable.Cols.SOLVED));
```

Using a CursorWrapper

- A **Cursor** give us raw column values.
- Pulling data out of a Cursor looks like this:

```
String uuidString = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.UUID));
String title = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.TITLE));
long date = cursor.getLong(
    cursor.getColumnIndex(CrimeTable.Cols.DATE));
int isSolved = cursor.getInt(
    cursor.getColumnIndex(CrimeTable.Cols.SOLVED));
```

Using a CursorWrapper

```
String uuidString = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.UUID));
String title = cursor.getString(
    cursor.getColumnIndex(CrimeTable.Cols.TITLE));
long date = cursor.getLong(
    cursor.getColumnIndex(CrimeTable.Cols.DATE));
int isSolved = cursor.getInt(
    cursor.getColumnIndex(CrimeTable.Cols.SOLVED));
```

- We need to repeat this code, every time we pull date out of a cursor
- Remember the **DRY** rule of thumb: **Don’t repeat yourself**.
- To read data from a Cursor, we can create Cursor subclass
- The easiest way to write a Cursor subclass is to use CursorWrapper.
- A CursorWrapper lets you wrap a Cursor you received from another place and add new methods on top of it.

Creating CrimeCursorWrapper (CrimeCursorWrapper.java)

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

- That creates a thin wrapper around a Cursor.
- It has all the same methods as the Cursor it wraps, and calling those methods does the exact same thing.

Adding getCrime() method (CrimeCursorWrapper.java)

- Now we can add our method to pull out relevant column data

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

    public Crime getCrime() {
        String uuidString = getString(getColumnIndex(CrimeTable.Cols.UUID));
        String title = getString(getColumnIndex(CrimeTable.Cols.TITLE));
        long date = getLong(getColumnIndex(CrimeTable.Cols.DATE));
        int isSolved = getInt(getColumnIndex(CrimeTable.Cols.SOLVED));

        return null;
    }
}
```

Adding getCrime() method (CrimeCursorWrapper.java)

- Now we can add our method to pull out relevant column data

```
public Crime getCrime() {
    String uuidString = getString(getColumnIndex(CrimeTable.Cols.UUID));
    String title = getString(getColumnIndex(CrimeTable.Cols.TITLE));
    long date = getLong(getColumnIndex(CrimeTable.Cols.DATE));
    int isSolved = getInt(getColumnIndex(CrimeTable.Cols.SOLVED));

    Crime crime = new Crime(UUID.fromString(uuidString));
    crime.setTitle(title);
    crime.setDate(new Date(date));
    crime.setSolved(isSolved != 0);

    return crime;
    ~~return null;~~
}
```

Converting to model objects

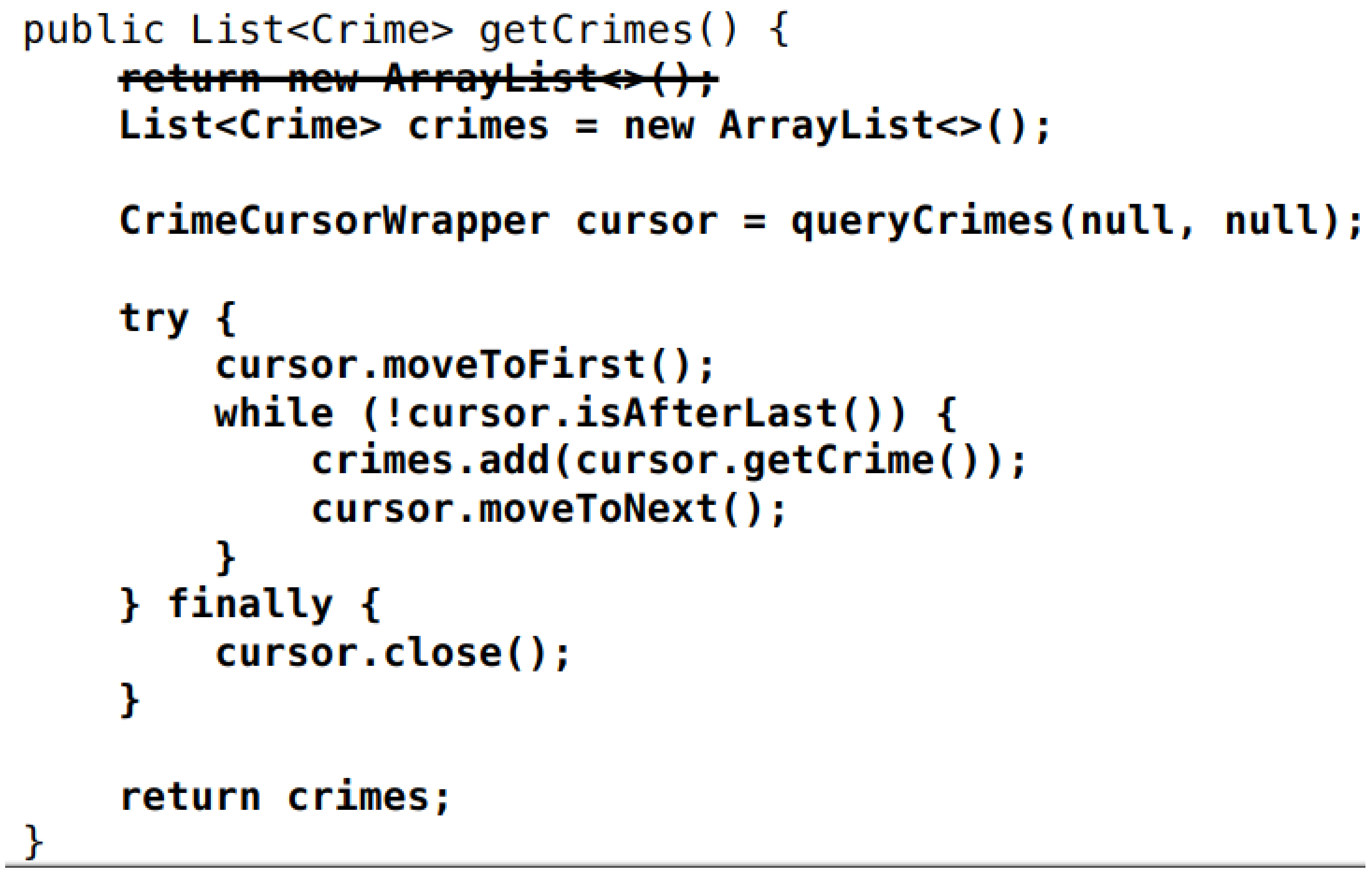
- With CrimeCursorWrapper, vending out a List<Crime> from CrimeLab will be straightforward.
- You need to wrap the cursor you get back from your query in a CrimeCursorWrapper, then iterate over it calling getCrime() to pull out its Crimes.

Update queryCrimes(…) to use CrimeCursorWrapper.

```
~~private Cursor queryCrimes(String whereClause, String[] whereArgs) {~~
private CrimeCursorWrapper queryCrimes(String whereClause, String[] whereArgs) {
    Cursor cursor = mDatabase.query(
            CrimeTable.NAME,
            null, // columns - null selects all columns
            whereClause,
            whereArgs,
            null, // groupBy
            null, // having
            null  // orderBy
    );

    ~~return cursor;~~
    return new CrimeCursorWrapper(cursor);
}
```

Returning crime list (CrimeLab.java)

- In getCrimes() method we will add code to query for all crimes, walk the cursor, and populate a Crime list.



```
public List<Crime> getCrimes() {
    ~~return new ArrayList<>();~~
    List<Crime> crimes = new ArrayList<>();

    CrimeCursorWrapper cursor = queryCrimes(null, null);

    try {
        cursor.moveToFirst();
        while (!cursor.isAfterLast()) {
            crimes.add(cursor.getCrime());
            cursor.moveToNext();
        }
    } finally {
        cursor.close();
    }

    return crimes;
}
```

Returning crime list (CrimeLab.java)

- Database cursors are called cursors because they always have their finger on a particular place in a query.
- So to pull the data out of a cursor, we move cursor to the first element by calling moveToFirst(), and then read in row data.
- To advance to a new row, we call moveToNext(), until finally isAfterLast() tells you that your pointer is off the end of the data set.
- The last **important** thing to do is to call **close()** on your Cursor.
- If we do not do it, we will eventually run out of open file handles and our app will crash

Rewriting getCrime(UUID) (CrimeLab.java)

```
public Crime getCrime(UUID id) {
    ~~return null;~~
    CrimeCursorWrapper cursor = queryCrimes(
            CrimeTable.Cols.UUID + " = ?",
            new String[] { id.toString() }
    );

    try {
        if (cursor.getCount() == 0) {
            return null;
        }

        cursor.moveToFirst();
        return cursor.getCrime();
    } finally {
        cursor.close();
    }
}
```

Refreshing model data

- Now the crimes are persistently stored to the database, but the persistent data is not read back in.
- The List<Crime> returned by getCrimes() is a snapshot of the Crimes at one point in time.
- To refresh CrimeListActivity, we need to update that snapshot.

Adding setCrimes(List<Crime>) (CrimeListFragment.java)

- We will add a setCrimes(List<Crime>) method to CrimeAdapter to swap out the crimes it displays.

```
private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {
    ...
    @Override
    public int getItemCount() {
        return mCrimes.size();
    }

    public void setCrimes(List<Crime> crimes) {
        mCrimes = crimes;
    }
}
```

Adding setCrimes(List<Crime>) (CrimeListFragment.java)

- We will add a setCrimes(List<Crime>) method to CrimeAdapter to swap out the crimes it displays.

```java
private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {
    ...
    @Override
    public int getItemCount() {
        return mCrimes.size();
    }

    public void setCrimes(List<Crime> crimes) {
        mCrimes = crimes;
    }
}
```

Calling setCrimes(List<>) (CrimeListFragment.java)

- Then we will call setCrimes(List<Crime>) in updateUI().

```
private void updateUI() {
    CrimeLab crimeLab = CrimeLab.get(getActivity());
    List<Crime> crimes = crimeLab.getCrimes();

    if (mAdapter == null) {
        mAdapter = new CrimeAdapter(crimes);
        mCrimeRecyclerView.setAdapter(mAdapter);
    } else {
        mAdapter.setCrimes(crimes);
        mAdapter.notifyDataSetChanged();
    }

    updateSubtitle();
}
```

InClass Task 15 (Deleting Crimes)

- If we added a Delete Crime action item earlier, this task builds off of that by adding the ability to delete crimes from our database by calling a deleteCrime(Crime) method on CrimeLab, which will call mDatabase.delete(…) to finish the job.

- And if you do not have a Delete Crime? Well, go ahead and add it! Add an action item to CrimeFragment’s toolbar that calls CrimeLab.deleteCrime(Crime) and finish()es its Activity.

Recommended Readings

- Page # 269 to 288, Chapter 14: SQLite Databases from Android Programming: The Big Nerd Ranch Guide, 3rd Edition by Bill Phillips, 2017