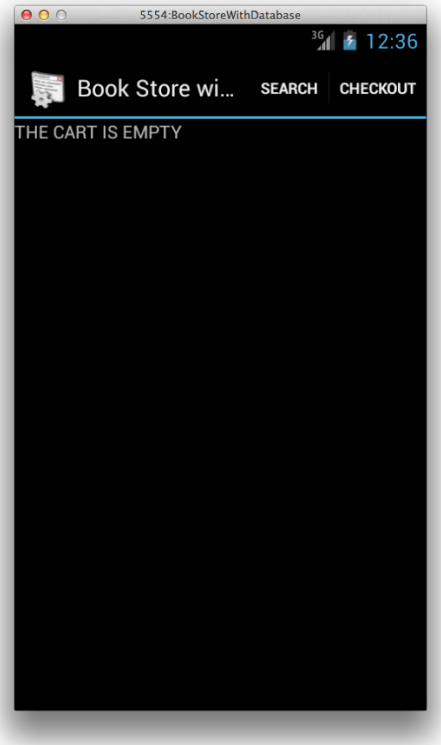
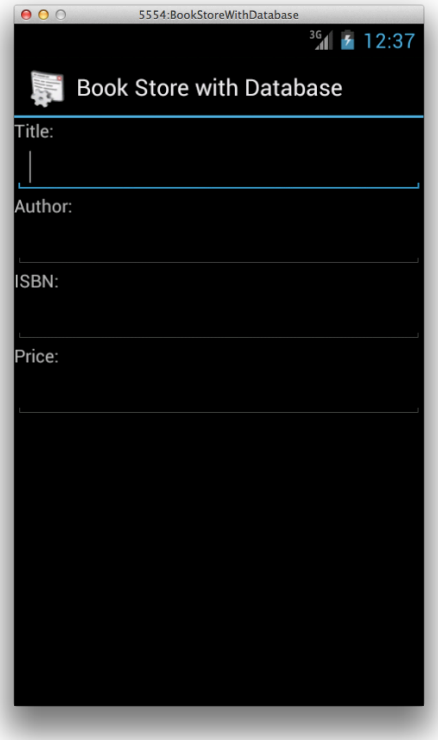
CS 522—Spring 2018
Mobile Systems and Applications
Assignment Three—Databases

Set your minimum SDK version to Lollipop (Android 5.1, API 22) for your submission for this assignment. Your app theme should extend `android:Theme.Material.Light.DarkActionBar`. We will be testing your app on a virtual Google Nexus 5X running API 22, so you should ensure that your app works on that platform, although you may develop on another device, e.g., your personal telephone.

Part 1: Book Store

In the previous assignment, you developed a book store app that saves the contents of a shopping cart in an in-memory array. For this assignment, you will modify the app to store the shopping cart in a SQLite database.





As before, in addition to the main activity, you will also have activities for adding a book, for viewing the details of a book, and for checking out. You can continue to return a new book entity object back to the main activity, and pass a book entity object to the activity for viewing a book details. However the shopping cart will now be persistent, and will not be destroyed if the app is exited. The "checkout" activity will clear the database. As before, you should have action bar items for the `Add` and `Checkout` activities, as well as handling the launching of sub-activities and the passing of data back and forth with intents.

You should follow these guidelines for your implementation.

First, define a subpackage of your application, called `contracts`. Define classes in this package called `BookContract` and `AuthorContract`, that define (as final static `String` constants) the names of the columns in your database. For each operation, define a type-specific read operation that reads the value of the column from a cursor, and a type-specific write operation that adds the value in a column to a `ContentValues` object. For example:

```
public static final String TITLE = “title”;

public static String getTitle(Cursor cursor) {
   return cursor.getString(cursor.getColumnIndexOrThrow(TITLE));
}

public static void putTitle(ContentValues values, String title) {
   values.put(TITLE, title);
}
```

Second, extend the `Book` and `Author` entity classes that you defined in the previous assignment, with a constructor that initializes the fields from an input cursor, and a method that writes the fields of the entity to a `ContentValues` object:

```
public class Book implements Parcelable {
   ...
   public Book(Cursor cursor) {
      this.title = BookContract.getTitle(cursor);
      ...
   }
   ...
   public void writeToProvider(ContentValues values) {
      BookContract.putTitle(values);
      ...
   }
}
```

In the `Author` class, define a constructor that takes a string for an author name and parses it as an `Author` object. This is used to extract a list of the authors for a book from a row in a book cursor. This is discussed more below.

Third, define a subpackage called `databases`. In this class, define the database adapter class `CartDbAdapter`. This class defines:

1. Useful string literals such as `DATABASE_CREATE` (the SQL command to create the database), `DATABASE_NAME` (the name of the file containing the database), `BOOK_TABLE` (the name of the table containing book information) and `AUTHOR_TABLE` (the name of the table containing author information), and `DATABASE_VERSION` (an integer that is used for versioning your database).

2. A private static inner class called `DatabaseHelper` that extends `SQLiteOpenHelper` with your logic for creating the database and upgrading where necessary. Your adapter class, on instantiation, instantiates this helper class in order to obtain a reference to the database.
3. Useful application-specific operations for accessing the database:

```
public void open() throws SQLException;
public Cursor fetchAllBooks();
public Book fetchBook(long rowId);
public void persist(Book book) throws SQLException;
public boolean delete(Book book);
public boolean deleteAll();
public void close();
```

Fourth, in your main activity where you display the contents of the shopping cart in a `ListView`, use the `SimpleCursorAdapter` class to connect the cursor resulting from a query to the list view. There are two constructors for `SimpleCursorAdapter`. It is okay **for this assignment only** to use the first, deprecated constructor, that performs cursor operations (such as requerying the database if the activity is restarted) on the main thread:

```
public SimpleCursorAdapter(Context context, int layout, Cursor c,
                           String[] from, int [] to);
```

You should call `startManagingCursor` in your activity to manage the cursor you get back from a query, through the activity life-cycle (closing and requerying). Again, this method is deprecated, and **you should only use it for this assignment**.

You can use the predefined layout resource `android.R.layout.simple_list_item_2` as the layout resource for each row in your list view. This defines a layout of two text views, one above the other, for each row. Display the title and authors of each book:

```
String[] from = new String[] { BookContract.TITLE,
                               BookContract.AUTHORS };
int[] to = new int[] { android.R.id.text1,
                       android.R.id.text2 };
```

One challenge with this app, which we may as well face head on, is handling multiple authors. For simplicity, you can assume that the relationship between books and authors is one-to-many rather than many-to-many.

```
CREATE TABLE Books (
   _id INTEGER PRIMARY KEY,
   ...
);
CREATE TABLE Authors (
   _id INTEGER PRIMARY KEY,
   ...
```

```
    book_fk INTEGER NOT NULL,
    FOREIGN KEY book_fk REFERENCES Books(_id) ON DELETE CASCADE
);
CREATE INDEX AuthorsBookIndex ON Authors(book_fk);
```

Note: You should **always** define table and column (and index) names just once as string literals, and then define a string expression that builds a SQL command such as above to create the database. You must define the secondary index on book foreign keys yourself, otherwise SQLite will perform linear searches of the database to enforce such constraints. By default, foreign key constraint enforcement is not enabled in SQLite. You can enable it *on each connection to the database* by executing:

```
db.execSQL("PRAGMA foreign_keys=ON;");
```

Once you have retrieved a book record, you could requery the database for the (N) authors, but this gives rise to the N+1 problem for ORM. It is particularly problematic if you have to do this for every book in a cursor. A better idea is to retrieve the author information with the book information, even though cursors don't support structured query results. Here is a way to do it. For inserting and retrieving book records, assume a synthetic column for the `Books` contract called `AUTHORS`, that is string-valued. This column is "synthetic" because it does not actually exist in the database, but the contract pretends that it does. A query returns a list of authors as a string, with a special separator character between the author names (Assume the parts of an author name are just separated by blanks):

```
public static final char SEPARATOR_CHAR = '|';

private static final Pattern SEPARATOR =
  Pattern.compile(Character.toString(SEPARATOR_CHAR), Pattern.LITERAL);

public static String[] readStringArray(String in) {
   return SEPARATOR.split(in);
}
```

To retrieve the information from the database, you will follow this strategy: perform a join on the book and author tables, on the book key field. Then group the results by book (each row will have the same values except for the author name), and aggregate the author names by concatenating them with the separator character:

```
SELECT Books._id, title, price, isbn, GROUP_CONCAT(name,'|') as authors
FROM Books JOIN Authors ON Books._id = Authors.book_fk
GROUP BY Books._id, title, price, isbn
```

Here I am assuming a single name field (called `name`) in the `Authors` table, and the synthetic column in a query result or insertion record is called `authors`. Again, always follow the best practice of defining these identifiers as string literals in your code, and then generate SQL expressions based on these. It may be preferable to use `LEFT OUTER`

`JOIN` as the join operator, because it allows books with no authors, but the `JOIN` operation is sufficient if you have problems with left outer join.

There is an issue with displaying author names in the list view. The user will see the names separated by the separator character. This is okay for now. We will see how to fix this in the next assignment, e.g., only showing the first author. Also, it is okay to assume that an author name is a single string, just so long as you continue to use a separate `Author` entity class for authors.

As with the previous assignment, you should also support the ability to delete a book that is selected from the list view. You should provide offer a contextual action bar (CAB) when a book is selected via a "long press" (at least two seconds). The CAB overrides the usual action bar while the item is selected, offering the "delete" option for the selected item in the action bar. Single-item selection (only being able to select a single book for deletion at a time) is sufficient for this assignment. An ordinary press should just cause the "view book" activity to be launched.

Part 2: Persistent Chat App

In this second part of the assignment, you will extend the chat server app from the previous assignment. The previous app just saves messages received in an array in the activity UI. If the user navigates away from the activity, and then returns to it, there is a good chance that the messages already received will have been lost, due to the way that Android manages resources in the activity life cycle. Another motivation is that we are so far violating one of the most basic tenets of Android programming, that no blocking operations should be performed on the main UI thread. In preparation for moving receipt of messages to background processing, we will in this assignment persist both messages received, and information about people that have sent us messages, to a database.

As with the first part of this assignment, you should define entity classes, a contract and a database adapter for your database. Define these in `entities`, `contracts` and `databases` subpackages, respectively, of your app. You have two kinds of entities: a message and a peer (message sender). We will assume a many-to-one relationship from messages to peers:

```
public class Message implements Parcelable {
   public long id;
   public String messageText;
   public Date timestamp;
   public String sender;
   public long senderId;
}

public class Peer implements Parcelable {
   public long id;
   public String name;
   public Date timestamp;
   public InetAddress address;
```

```
    public int port;
}
```

All of these entity objects should implement `Parcelable`, including a constructor that takes a parcel as its input argument, as well as a constructor for initializing from a cursor and an operation for writing to a `ContentValues` object.

Note: When constructing an `InetAddress` object, you should avoid the operation `getByName(String addr)`, since this will cause a blocking network operation as it performs a reverse DNS lookup. Instead you can use an operation provided by the Guava library:

```
import com.google.guava.InetAddresses;
InetAddress address = InetAddresses.forString(...);
```

A dependency on Guava is included in the project provided for this assignment. See the `InetAddressUtils` class in the project you are provided with.

When you receive a message, extract the name of the sender and where it came from (part of the UDP packet header information), and update the database record for this peer, or insert a record if this is the first time we have heard from this peer. A message record contains a foreign key reference to the peer record for the sender:

```
CREATE TABLE Peers (
    _id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    name LONG NOT NULL,
    address TEXT NOT NULL,
    ...
);
CREATE TABLE Messages (
    _id INTEGER PRIMARY KEY,
    ...
    peer_fk INTEGER NOT NULL,
    FOREIGN KEY peer_fk REFERENCES Peers(_id) ON DELETE CASCADE
);
CREATE INDEX MessagesPeerIndex ON Messages(peer_fk);
CREATE INDEX PeerNameIndex ON Peers(name);
```

Querying is simpler than in the previous part of the assignment. If you redundantly store a peer's user name along with the foreign key in a message record, you do not need to do a join when querying for a list of messages and their senders. The index on peer names is required for searching to see if a peer is already in the database, when a message from that peer is received.

In addition, provide another UI (accessible using the options menu from the action bar) for displaying a list of the peers. This UI just lists peers by name in a list view.

If the user selects one of these peers, then provide in a third UI information about that peer (their currently known IP address and port, and the last time that a message was received from that user). Pass the primary key for the peer to the subactivity, that will then query the database for the information to be displayed.

To support all of this, do the following:

1. Your contract should define column names for the attributes stored in the database, and getter and putter methods (for cursors and content values, respectively).
2. Your entity classes should define constructors with cursors as arguments, and `writeToProvider` operations for writing to `ContentValue` objects.
3. Your database adapter should provide all the operations your application needs to interact with the database without explicitly handling cursors and content values. The only exception to this is where you are returning a cursor to be displayed in a list view (which all your activities do).

Submitting Your Assignment

Once you have your code working, please follow these instructions for submitting your assignment:

1. Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory Humphrey_Bogart.
2. In that directory you should provide the Android Studio projects for your apps.

In addition, record short flash, mpeg, avi or Quicktime videos of your apps working. Make sure that your name appears at the beginning of the videos. For example, put your name in the title of the app. *Do not provide private information such as your email or cwid in the video*. The videos should demonstrate running the app, exiting the app, then rerunning the app and demonstrating that the saved state from the previous run (shopping cart contents for the Bookstore app, client name, list of peers and messages for the chat app) are still present when the app is restarted.

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have three Android projects, for the apps you have built. You should also provide videos demonstrating the working of your assignments