Mobile Application Lab 8

Instructors: Legand Burge, Asad Ullah Naweed
Date: 09/18/2018
Topics: DialogFragments and SQLite Databases

Be sure to click on the links in this doc to view more information about certain topics. These may or may not help you in your lab, but can always provide some extra context and information.

Initial Setup

You'll be creating a simple contacts saving app for this lab. Be sure to start off with a clean project. You dont need to bother with fragments for this one; just use an Activity.

Basic UI Setup

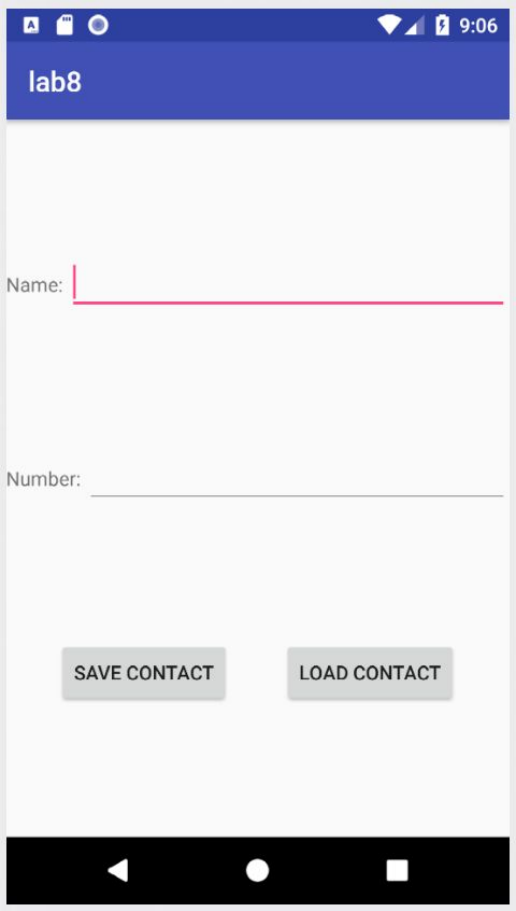
The layout for your `MainActivity` is given below. Copy-paste it into your project as it is.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:id="@+id/name_layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintBottom_toTopOf="@id/number_layout"
        app:layout_constraintRight_toRightOf="parent" >
        <TextView
            android:text="Name: "
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
```

```
        <EditText
            android:id="@+id/name_edit_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1" />
    </LinearLayout>

    <LinearLayout
        android:id="@+id/number_layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintTop_toBottomOf="@id/name_layout"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintBottom_toTopOf="@id/save_button"
        app:layout_constraintRight_toRightOf="parent" >
        <TextView
            android:text="Number: "
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
        <EditText
            android:id="@+id/number_edit_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1" />
    </LinearLayout>

    <Button
        android:id="@+id/save_button"
        android:text="Save Contact"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/number_layout"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toLeftOf="@id/load_button"
        app:layout_constraintBottom_toBottomOf="parent"/>

    <Button
        android:id="@+id/load_button"
        android:text="Load Contact"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/number_layout"
        app:layout_constraintLeft_toRightOf="@id/save_button"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"/>
```

```
</android.support.constraint.ConstraintLayout>
```

Build and run your app. Make sure you see a screen like this:



Once this is building as it is, be sure to add two new private methods to your `MainActivity` class called `onSaveButtonPressed` and `onLoadButtonPressed`. Set these to be the callbacks invoked for the "Save" and "Load" buttons respectively.

The functionality to be implemented for this app is very simple. When the save button is pressed, the entered name and number need to be saved. When the load button is pressed, the number for the entered name needs to be loaded and set in the number field.

Saving 1 - SharedPreferences

Add the following code to your `MainActivity` class:

```
public class MainActivity extends AppCompatActivity {
  private static final String SHARED_PREFS_FILE_NAME = "Contacts_SharedPrefs";

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button saveButton = findViewById(R.id.save_button);
    saveButton.setOnClickListener(v -> onSaveButtonPressed());
    Button loadButton = findViewById(R.id.load_button);
    loadButton.setOnClickListener(v -> onLoadButtonPressed());
  }

  private void onSaveButtonPressed() {
    SharedPreferences sharedPrefs = getSharedPreferences(SHARED_PREFS_FILE_NAME,
        Context.MODE_PRIVATE);
  }

  private void onLoadButtonPressed() {
    SharedPreferences sharedPrefs = getSharedPreferences(SHARED_PREFS_FILE_NAME,
        Context.MODE_PRIVATE);
  }
}
```

Look at what this is doing. You now have a handle to the SharedPreferences object to read/write values to the device's private storage.

Now, write the number to the sharedPreferences using the name as the key. Recall and reuse the code from your last lab to do this.

Saving 2 - SQLite Database

For this simple application, we saw that it was very easy to implement it using SharedPreferences. However, that simply does not scale when the number of records increases into the thousands. We're now going to see how we can save and load this data to a local SQLite database.

A SQLite database, just like a SQL database, is a relational database. Each database has one or more tables, and each table has a set of columns. Each row in a table is called a record.

SQLite databases are backed by files present in the application's private space, so they have the same protections as SharedPreference's open in `MODE_PRIVATE`.

There's going to be slightly more code to write for SQLite databases, but it will be much more scalable when dealing with medium to large amounts of data.

Create the SQL Database Schema

It is good practice to DRY out your code. DRY stands for "Don't Repeat Yourself" and is a good principle to follow when writing a program: If you write something down, write it down in a single authoritative place and reference it everywhere else instead of repeating yourself.

In the spirit of DRY, we will create a `ContactDbSchema` class to hold a few constants:

```
public class ContactDbSchema {
  public static final String TABLE_NAME = "contacts";

  public static final class Cols {
    public static final String NAME = "name";
    public static final String NUMBER = "number";
  }
}
```

This class has no instance variables, and is simply a place to hold hard-coded DB schema related constants which we will need to refer to over and over.

Create the SQL Helper

It's much easier to deal with SQLite Databases with an extension of `SQLiteOpenHelper`.

```
class ContactSqliteHelper extends SQLiteOpenHelper {
  private static final String DB_NAME = "TechExchange_Contacts";
  private static final int VERSION = 1;

  ContactSqliteHelper(Context context) {
    super(context, DB_NAME, null, VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL("create table " + ContactDbSchema.TABLE_NAME +
        "( _id integer primary key autoincrement, " +
        ContactDbSchema.Cols.NAME + ", " +
        ContactDbSchema.Cols.NUMBER + ")");
  }

  @Override
```

```
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  }
}
```

This class is not that complicated. It is a helper class that provides a hook for creating a database if one does not exist already. We won't have to deal with the `onUpgrade` method today, so feel free to leave it blank.

For those of you who have seen SQL databases before, this should look very familiar. We will see moving forward that the SQLite API is actually much simpler than the full-fledged SQL language, even though you can write raw queries as well. One major thing to note here is that the column data types do not need to be specified when creating a database in SQLite. It's still advisable to do it though, even though we're avoiding it here for the sake of brevity.

Now that we have our helper, we can use it in `MainActivity` as follows:

```
public class MainActivity extends AppCompatActivity {
  private static final String SHARED_PREFS_FILE_NAME = "Contacts_SharedPrefs";

  private SQLiteDatabase database;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Button saveButton = findViewById(R.id.save_button);
    saveButton.setOnClickListener(v -> onSaveButtonPressed());
    Button loadButton = findViewById(R.id.load_button);
    loadButton.setOnClickListener(v -> onLoadButtonPressed());

    database = new ContactSqliteHelper(this).getWritableDatabase();
  }

  private void onSaveButtonPressed() {
  }

  private void onLoadButtonPressed() {
  }
}
```

Now, during save, we need to either do an "insert" or an "update". An insert operation inserts a brand new record into the database, whereas an update operation simply updates an existing one. To check if we need to do an insert or an update, we will need to first query the database

to check for the existence of any records with the same name.

Now, take a look at the documentation for `SQLiteDatabase.query()`. There are many overloaded versions of that method, but you should look at the one with the signature:

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

Open the documentation for this particular version and read the documentation. This is a method that you can call with specific arguments without having to compose a long SQL query yourself. It's also much safer than composing your own query: it guards the database against SQL injection attacks.

Implementing the queries for Save

You will now implement these queries, and then use the Cursor to iterate over the results. Copy-paste the following into your `onSaveButtonPressed` method:

```
Cursor cursor = database.query(
    ContactDbSchema.TABLE_NAME,
    null,
    ContactDbSchema.Cols.NAME + " = ?",
    new String[]{nameText},
    null, null, null);
```

Now, take a look at what is being passed for each query, and what the arguments represent. Pay close attention to the `selection` and `selectionArgs` arguments. These will create the WHERE clause in the query, with the elements in `selectionArgs` safely replacing the "?" signs in the `selection` string.

You now have a Cursor to iterate over the results. In this case, we only want to check if there are any results at all, or if there were no records retrieved for that query. Take a look at the Cursor documentation, and look at the `isAfterLast` method. The Cursor as an iterator that goes over each record one by one. When there are no more records, the cursor points to the position "after" the last element in the result set.

The Cursor also needs to be closed once you're done with it. The cleanest way to do this would be using the try-with-resources statement. Add the following lines of code to in your `MainActivity` file:

```
private void onSaveButtonPressed() {
  try (Cursor cursor = database.query(
        ContactDbSchema.TABLE_NAME,
        null,
        ContactDbSchema.Cols.NAME + " = ?",
        new String[]{nameText},
        null, null, null)) {
    // Make the cursor point to the first row in the result set.
    cursor.moveToFirst();
    if (cursor.isAfterLast()) {
      // No records exist. Need to insert a new one.
    } else {
      // A record already exists. Need to update it with the new number.
    }
  }
}
```

This is really convenient. It scopes the cursor within the try block, and is guaranteed to close the cursor no matter what errors happen inside the try-block.

Now, implement both the if-block and its corresponding else-block as an exercise. Be sure to read the documentation for the `insert` and `update` methods in SQLiteDatabase.

Also, be sure to guard against errors that could happen when the fields are empty. You should actively check for those, and display an appropriate `Toast` message if the user entered some bad input.

Implementing the queries for Load

Now that you're familiar with how to create a cursor, you will now need to use it to actually retrieve the results of a query.

Take a look at the documentation for Cursor again. Notice the `getString` method. It takes in a single integer argument called `columnIndex`. This method can be used to get a String value from the specified column index of the result set that the cursor is pointing to. To get the column index for a particular column name, use the `getColumnIndex` method.

Your `onLoadButtonPressed` should now look like this:

```
private void onLoadButtonPressed() {
  try (Cursor cursor = database.query( ... )) {
    cursor.moveToFirst();
    if (cursor.isAfterLast()) {
```

```
      // The contact was not found.
    }
    int colIndex = cursor.getColumnIndex(ContactDbSchema.Cols.NUMBER);
    if (colIndex < 0) {
      // The column was not found in the cursor.
    }
    String numberStr = cursor.getString(colIndex);
  }
}
```

There are a few things you need to do here. First, figure out what the query should be. Next, you need to write code to display `Toast` messages if the contact was not found, or if the query was bad. Then, you need to update the UI to set the Number that was retrieved.