Android: Room Persistence Library Exercises

Based on the [Android Room with a View](#) codelab.

BACK UP YOUR PROJECT BEFORE PROCEEDING

Update gradle files

Add the following code to your *build.gradle* (Module: app) file, at the end of the dependencies block.

```
// Room components
implementation "android.arch.persistence.room:runtime:$rootProject.roomVersion"
annotationProcessor "android.arch.persistence.room:compiler:$rootProject.roomVersion"
androidTestImplementation "android.arch.persistence.room:testing:$rootProject.roomVersion"

// Lifecycle components
implementation "android.arch.lifecycle:extensions:$rootProject.archLifecycleVersion"
annotationProcessor "android.arch.lifecycle:compiler:$rootProject.archLifecycleVersion"
```

In your *build.grade* (Project: HelloWorld), add the version numbers to the end of the file

```
ext {
    roomVersion = '1.1.1'
    archLifecycleVersion = '1.1.1'
}
```

See [Adding Components to your Project](#) for the most current version numbers.

Create the entity

Turn the class Contact into an entity by editing it to look as follows:

```
@Entity
public class Contact {
    @PrimaryKey(autoGenerate = true)
    public long id;
    public String name;
    public String email;
    public String mobile;

    ...
}
```

Create the Data Access Object

Create a new interface called `ContactDao` in the package `comp575.helloworld` with the following declaration:

```
@Dao
public interface ContactDao {
    @Insert
    void insert(Contact contact);

    @Update
    void update(Contact contact);

    @Delete
    void delete(Contact contact);

    @Query("SELECT * from Contact ORDER BY name")
    LiveData<List<Contact>> getAllContacts();
}
```

The data access object is used to specify SQL queries and associate them with method calls. The compiler checks the SQL and generates queries from convenience annotations.

LiveData causes Room to generate code to update the LiveData when the database is updated. An Observer can be registered to receive notifications when the data changes.

Add a Room database

Room is a database layer on top of an SQLite database. Room takes care of mundane tasks that used to be handled with an SQLiteOpenHelper.

Room uses the DAO to issue queries to its database.

By default, to avoid poor UI performance, Room doesn't allow database queries to be issued on the main thread. LiveData avoids this by running the query asynchronously on a background thread.

Room provides compile-time checks of SQLite statements.

Create a new class called `ContactRoomDatabase` in the package `comp575.helloworld` with the following declaration:

```
@Database(entities = {Contact.class}, version = 1)
public abstract class ContactRoomDatabase extends RoomDatabase {
    public abstract ContactDao contactDao();

    private static ContactRoomDatabase INSTANCE;

    public static synchronized ContactRoomDatabase getDatabase(Context context) {
        if (INSTANCE == null) {
            INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                    ContactRoomDatabase.class, "contact_database").build();
        }
        return INSTANCE;
    }
}
```

The last argument of `Room.databaseBuilder` is the name of the database file.

Create the Repository

The Repository is not part of the Room framework, but is a suggested best practice for code separation and architecture. A Repository class handles data operations. It provides a clean API to the rest of the app for app data.

Create a new class called `ContactRepository` in the package `comp575.helloworld` with the following declaration:

```
public class ContactRepository {
   private ContactDao contactDao;
   private LiveData<List<Contact>> allContacts;

   ContactRepository(Context context) {
       ContactRoomDatabase db = ContactRoomDatabase.getDatabase(context);
       contactDao = db.contactDao();
       allContacts = contactDao.getAllContacts();
   }

   public void insert(Contact contact) {
       new InsertAsyncTask().execute(contact);
   }

   private class InsertAsyncTask extends AsyncTask<Contact, Void, Void> {
       @Override
       protected Void doInBackground(final Contact... params) {
           for (Contact contact : params) {
               contactDao.insert(contact);
           }
           return null;
       }
   }

   public void update(Contact contact) {
       new UpdateAsyncTask().execute(contact);
   }

   private class UpdateAsyncTask extends AsyncTask<Contact, Void, Void> {
       @Override
       protected Void doInBackground(final Contact... params) {
           for (Contact contact : params) {
               contactDao.update(contact);
           }
           return null;
       }
   }

   public LiveData<List<Contact>> getAllContacts() {
       return allContacts;
   }
}
```

Connect with the data

Add a *ContactRepository* instance variable to *MainActivity*:

```
public class MainActivity ... {
    ...
    private ContactRepository contactRepository;
```

In the method *onCreate* of *MainActivity*, create a *ContactRepository* and register an observer:

```
contactRepository = new ContactRepository(this);
contactRepository.getAllContacts().observe(this, new Observer<List<Contact>>() {
    @Override
    public void onChanged(List<Contact> updatedContacts) {
        // update the contacts list when the database changes
        adapter.clear();
        adapter.addAll(updatedContacts);
    }
});
```

Keep the code that uses *savedInstanceState* for assessment purposes, but comment out the lines that add Joe Bloggs and Jane Doe to the contacts.

Change the method *saveContact* of *MainActivity* to modify the database rather than the ArrayList.

- **It should still look up the contact in the ArrayList by name.**
- **If a contact with that name does not exist, the contact should be inserted using `contactRepository.insert(...)`. Comment out the old code that adds the contact the ArrayList.**
- **If a contact with that name already exists, the contact should be updated. Set the *email* and *mobile* fields of the existing contact then call `contactRepository.update(...)` and pass it the updated contact. Comment out any old code that updates the ArrayList.**

Deleting Contacts

Add a button next to the "Save Contact" button that deletes the currently displayed contact.

You will need to edit the layout, write a method in the MainActivity class which will run when the new button is clicked, and write a method that deletes contacts in the ContactRepository class.