Name:

**CSSE490 Android Application Development**



**Lab 7: Rose Homework**

The purpose of this lab is to get you familiar with typical SQLite database functionality, as well as to give you some practice in making your applications persist data correctly.

**Part A: Data Model**

**Part B: Viewing the Data**

**Part C: Adding, Editing, and Deleting Data**

**Part D: Using the Database**

**Part E: Displaying Persistence**

In order to check off this lab, you can demo the final part.

In this lab you will create a lightweight to­do list application for homework assignments. The application will allow users to enter new assignments, specifying the date each one is due (required), as well as a homework assignment name (required). The main application view will display this list of items to the user, ordering them by due date (soonest first), then name (alphabetical, case insensitive). Users will be able to tap any item on the list to delete it once they finish it.

The goal is to learn about using SQLite for data persistence. This app may need more features to be truly useful, but more features dilute the SQLite focus.

The lab is written with lots of hints built into it. We recommend trying to do each section on your own for practice before looking at the given code.

One more note: in the lab, we take a slightly different approach to some things than we did in lecture, just to show you some more alternatives.

**Part A: Create Project, Data Model**

Start by making a new Android project in Eclipse called “RoseHomework” with an Activity called

“DisplayTasksActivity” that extends “android.app.ListActivity”:

Using the MVC paradigm, we’ll first want to define what a “task” is. Create a new class in your package called Task (use the default parameters in the wizard, just inheriting from java.lang.Object).

We want our task to have 3 attributes:

● Name

● Course

● Date due

It should also have an ID field to keep track of which task is which in the database, even if the names or descriptions change.

It should implement Comparable so we can easily sort lists of Tasks, which will be handy later. Your toString() method should display more information, like the due date of the task.

We’ll use this class as our primary data object, both for displaying in views and passing to the database to be saved, edited, and deleted. This effectively forms our entire model layer, in terms of MVC.

**package** edu.rosehulman.rosehomework;

**import** java.util.Calendar;

**import** java.util.GregorianCalendar;

**public class** Task **implements** Comparable<Task> {

**private long** mId; **private** String mName; **private** String mCourse;

**private** GregorianCalendar mDueDate;

**public static final** String[] *MONTHS* = { "JAN", "FEB", "MAR", "APR", "MAY", "JUN", "JUL", "AUG", "SEP", "OCT", "NOV", "DEC" };

**public** Task() { /\* no-op \*/ }

**public** Task(String name, String course, GregorianCalendar dueDate) {

mName = name; mCourse = course; mDueDate = dueDate; mId = -1;

}

**public long** getId() { **return** mId; }

**public void** setId(**long** id) { **this**.mId = id; }

**public** String getName() { **return** mName; }

**public void** setName(String name) { **this**.mName = name; }

**public** String getCourse() { **return** mCourse; }

**public void** setCourse(String course) { **this**.mCourse = course; } **public** GregorianCalendar getDueDate() { **return** mDueDate; } **public int** getYearDue() { **return** mDueDate.get(Calendar.*YEAR*); }

**public int** getMonthDue() { **return** mDueDate.get(Calendar.*MONTH*); }

**public int** getDayOfMonthDue() { **return** mDueDate.get(Calendar.*DAY\_OF\_MONTH*); } **public void** setDueDate(GregorianCalendar dueDate) { **this**.mDueDate = dueDate; } **public void** setDueDate(**int** year, **int** month, **int** dayOfMonth) {

mDueDate = **new** GregorianCalendar();

mDueDate.set(Calendar.*HOUR*, 0); mDueDate.set(Calendar.*MINUTE*, 0); mDueDate.set(Calendar.*SECOND*, 0); mDueDate.set(Calendar.*MILLISECOND*, 0); mDueDate.set(Calendar.*YEAR*, year); mDueDate.set(Calendar.*MONTH*, month + 1); mDueDate.set(Calendar.*DAY\_OF\_MONTH*, dayOfMonth);

}

@Override

**public int** compareTo(Task another) {

GregorianCalendar anotherDueDate = another.getDueDate();

**int** comp = mDueDate.compareTo(anotherDueDate);

**return** (comp == 0 ? mName.compareTo(another.getName()) : comp);

}

@Override

**public** String toString() {

**return** *MONTHS*[**this**.mDueDate.get(Calendar.*MONTH*) - 1] + " "

+ **this**.mDueDate.get(Calendar.*DAY\_OF\_MONTH*) + ", "

+ **this**.mDueDate.get(Calendar.*YEAR*) + " " + mCourse + " "

+ mName;

}

}

----

package edu.rosehulman.rosehomework;

import java.util.Calendar;

import java.util.GregorianCalendar;

public class Task implements Comparable<Task> {

private long mId;

private String mName;

private String mCourse;

private GregorianCalendar mDueDate;

public static final String[] MONTHS = { "JAN", "FEB", "MAR", "APR", "MAY",

"JUN", "JUL", "AUG", "SEP", "OCT", "NOV", "DEC" };

public Task() { }

public Task(String name, String course, GregorianCalendar date) {

this.mName = name;

this.mCourse = course;

this.mDueDate = date;

}

public long getId() { return mId;}

public void setId(long id) { this.mId = id; }

public String getName() { return mName;}

public void setName(String name) { this.mName = name; }

public String getCourse() { return mCourse; }

public void setCourse(String mCourse) { this.mCourse = mCourse; }

public GregorianCalendar getDueDate() { return mDueDate; }

public int getYearDue() { return getDueDate().get(Calendar.YEAR); }

public int getMonthDue() { return getDueDate().get(Calendar.MONTH); }

public int getDayOfMonthDue() { return getDueDate().get(Calendar.DAY\_OF\_MONTH); }

public void setDueDate(GregorianCalendar date) { this.mDueDate = date; }

public void setDateDue(int year, int month, int dayOfMonth) { mDueDate = new GregorianCalendar(); mDueDate.set(Calendar.HOUR, 0); mDueDate.set(Calendar.MINUTE, 0); mDueDate.set(Calendar.SECOND, 0); mDueDate.set(Calendar.MILLISECOND, 0);

mDueDate.set(Calendar.YEAR, year); mDueDate.set(Calendar.MONTH, month + 1); mDueDate.set(Calendar.DAY\_OF\_MONTH, dayOfMonth);

}

public String toString() {

return MONTHS[this.mDueDate.get(Calendar.MONTH) - 1] + " "

+ this.mDueDate.get(Calendar.DAY\_OF\_MONTH) + ", "

+ this.mDueDate.get(Calendar.YEAR) + " " + this.getCourse()

+ " " + this.getName();

}

public int compareTo(Task another) {

GregorianCalendar otherCalendar = another.getDueDate();

if (otherCalendar.compareTo(getDueDate()) == 0)

return getName().compareTo(another.getName());

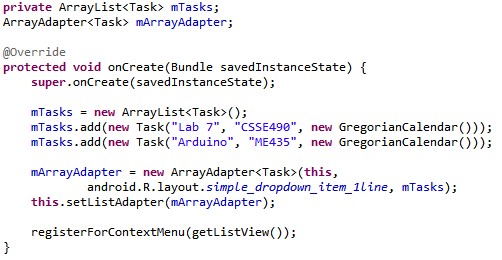
return -otherCalendar.compareTo(getDueDate());

}

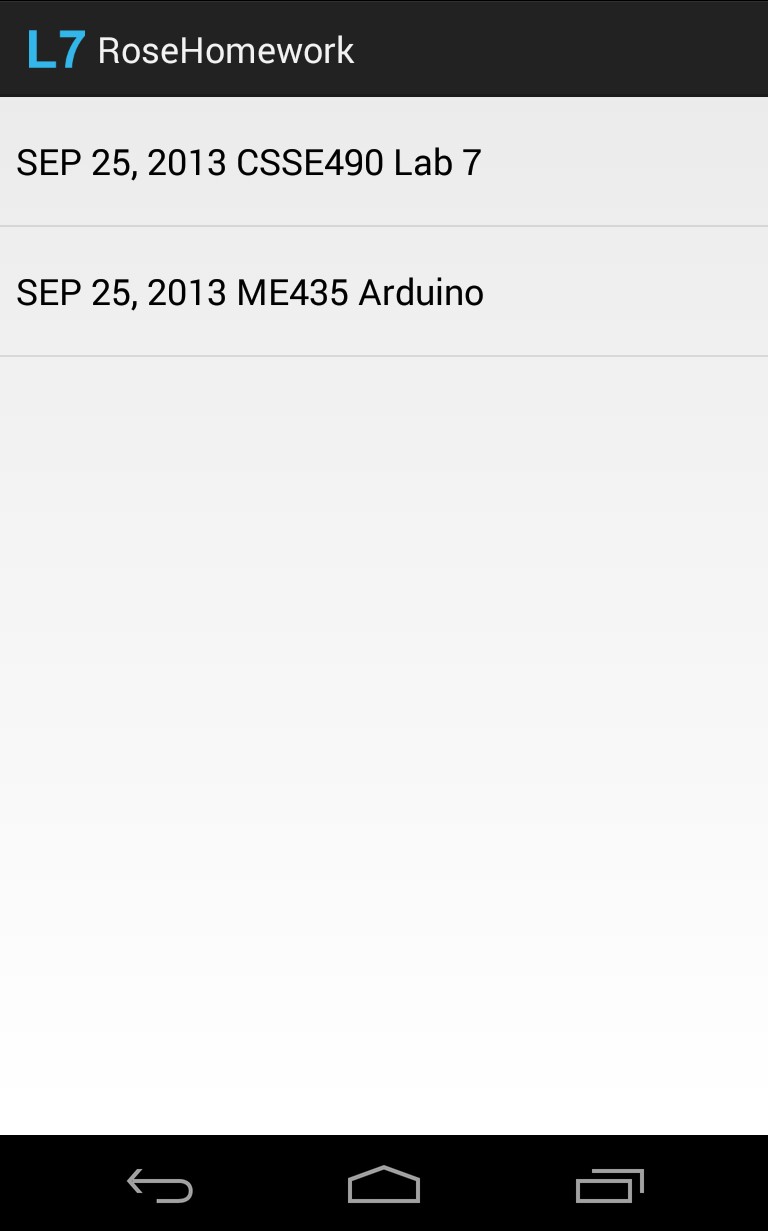
}

**Part B: Viewing Task List**

In your main activity, **DisplayTasks**, connect your ListView to an ArrayAdapter of Tasks, for now filling it with dummy values.



You should end up with something like this.

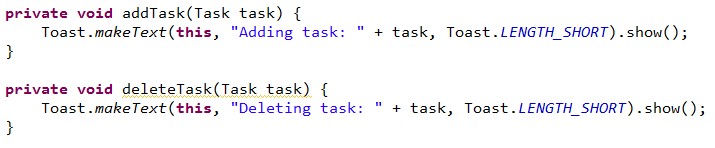


The viewing portion of this Activity is now more or less finished; we just need to use the database to populate the list instead of sticking values in there ourselves.

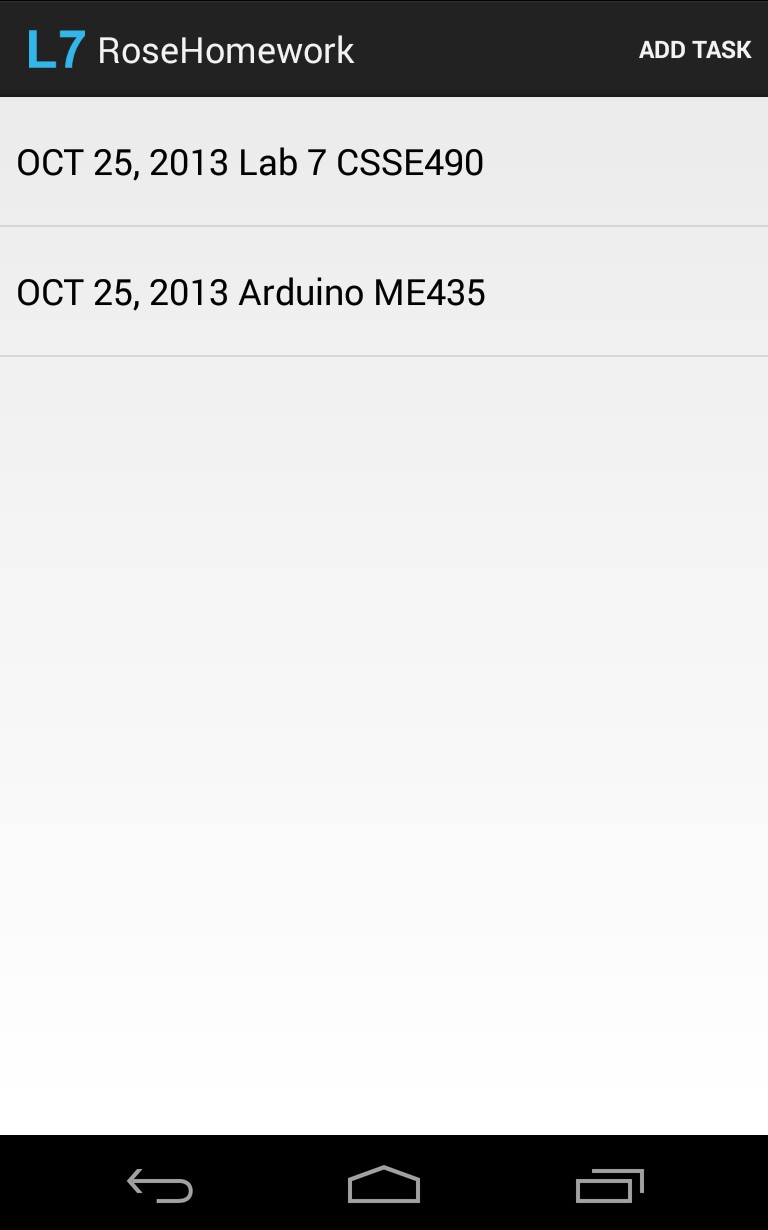
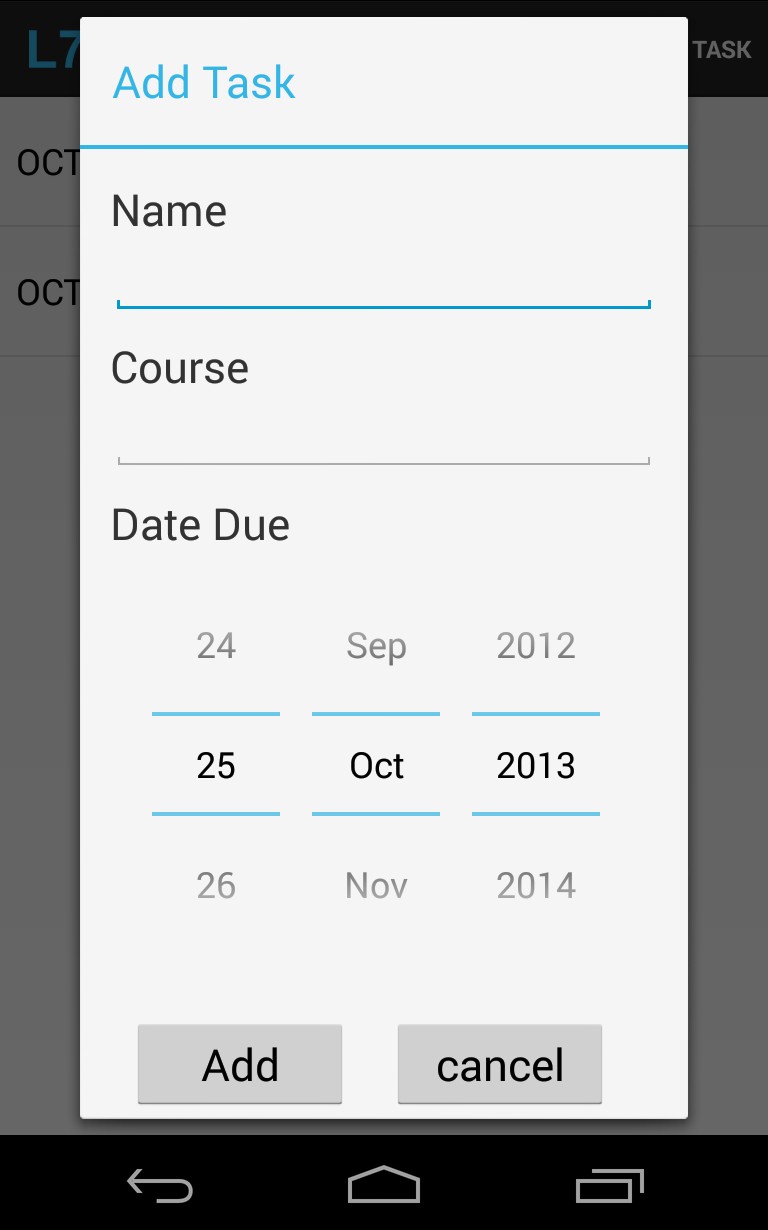
**Part C: Adding and Deleting Tasks**

We’re now going to add mechanisms for adding and deleting data. We won’t do an edit screen for the sake of time. We want to spend more time focusing on the database functionality.

We’ll first need to stub out two methods in our DisplayTasks Activity that will eventually contain database calls.



We’ll create a menu item button that will call a custom dialog to create new tasks.



Define the dialog layout in a new layout file, **create\_dialog.xml**.

<?xml version="1.0" encoding="utf-8"?>

<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" >

<LinearLayout android:layout\_width="match\_parent" android:layout\_height="wrap\_content" android:orientation="vertical" android:padding="15dp" >

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"

android:text="@string/name" android:textSize="22sp" />

<EditText android:id="@+id/create\_dialog\_name" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:layout\_marginBottom="10dp" >

<requestFocus />

</EditText>

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@string/course" android:textSize="22sp" />

<EditText android:id="@+id/create\_dialog\_course" android:layout\_width="fill\_parent" android:layout\_height="wrap\_content" android:layout\_marginBottom="10dp" >

</EditText>

<TextView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:text="@string/date\_due" android:textSize="22sp" />

<DatePicker android:id="@+id/create\_dialog\_date" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_gravity="center\_horizontal" android:layout\_marginBottom="10dp" android:calendarViewShown="false"/>

<LinearLayout android:layout\_width="match\_parent" android:layout\_height="wrap\_content" android:layout\_gravity="center\_horizontal" >

<Button android:id="@+id/create\_dialog\_add\_button"

android:layout\_width="110dp" android:layout\_height="wrap\_content" android:layout\_margin="10dp" android:text="@string/add" android:textSize="22sp" />

<Button android:id="@+id/create\_dialog\_cancel\_button" android:layout\_width="110dp" android:layout\_height="wrap\_content" android:layout\_margin="10dp" android:text="@string/cancel" android:textSize="22sp" />

</LinearLayout>

</LinearLayout>

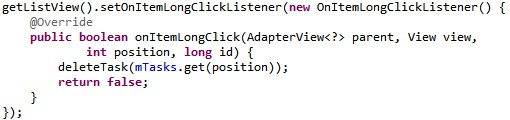
</ScrollView>

(update this?)

Now create a method that creates a dialog that uses the layout you just defined. If the user presses the add button, the dialog should call the addTask() method to add the task. Have the “Add Task” menu Item call this method to start the dialog. You should be comfortable writing Dialogs by now, so you are on your own here.

We will long­press tasks to delete them, so in the OnCreate method set up an

OnItemLongClickListener for the ListView:

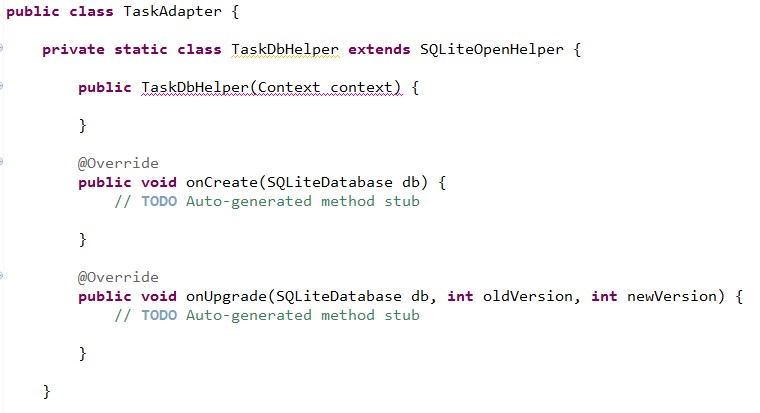


When you run the application, you should now be able to use the application menu to open the task creation dialog, as shown above.

You should notice that adding a task and long pressing one to delete it don’t do anything except display a toast. This is because we’re just making calls to the methods we stubbed out at the beginning of this section. In the next section, we’ll create a SQLite database and actually add and remove Tasks from it.

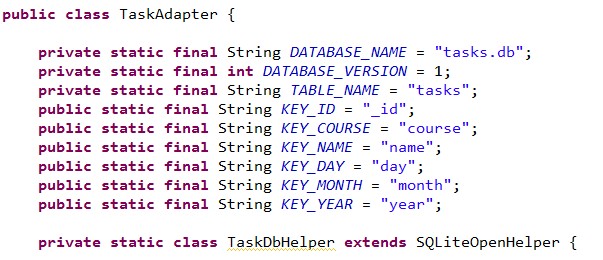
**Part D: Using the Database**

On to hooking our application into a SQLite database. To do this, make a new class in the package called  **TaskDataAdapter**, but also give it a private inner class, **TaskDbHelper**, that extends SQLiteOpenHelper.

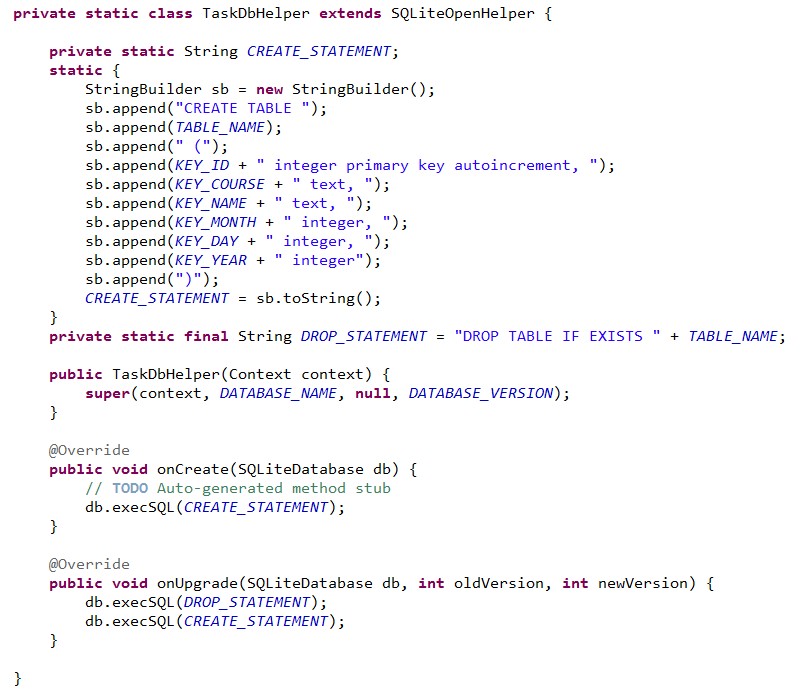


At present the compiler complains about the empty constructor. That’s expected.

Let’s define some constants in our main class, including our database name (the name of the file), our version (1 for now, but it should change whenever we change the schema), the name of the one table we’ll need to create, and the names and indices of our columns in that table.

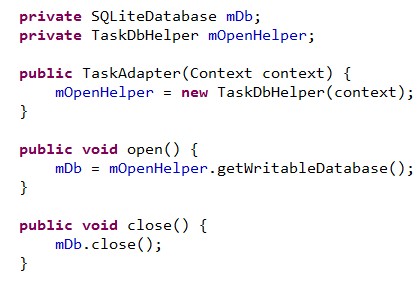


Now let’s define the SQL statements that will execute when the database is created and upgraded, then execute them in the correct places. We’ll also fill in the rest of our helper class.



The helper class is now done.

Next we’ll need a constructor and open and close methods on our adapter.



We’ll now continue writing more of the TaskAdapter.

We can focus our application­specific methods for moving data into and out of the database. For now, we’ll just need to cover two basic table operations:

● Insert a new Task

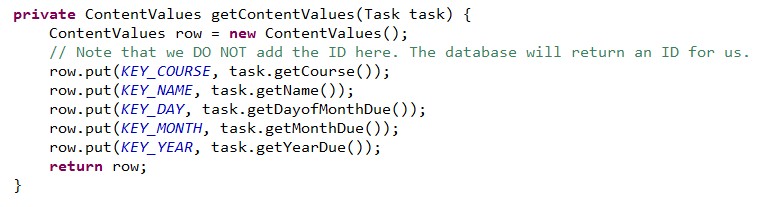
○ Then find out its Id and return a Task with the Id set to the returned Id.

● Delete a Task

○ Based on the Task Id

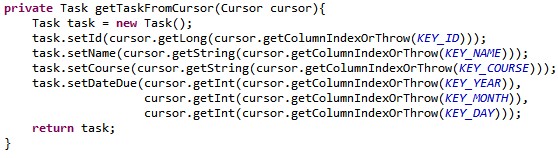
Let’s look at inserting a Task first. We’ll be doing two separate operations here: adding the data to the database, which will create a new row in our table, then retrieving that row to get the proper Id value to populate the Task object that we’ll return.

Our first step is to write a helper method that will convert a Task to a ContentValues object, which we’ll use to insert into the database. Note that we **do not** add the id here.

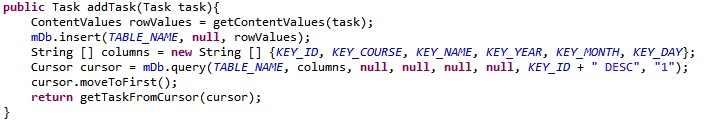


We’ll also need to be able to create a Task object from the results of a query, which are

delivered to us in the form of a Cursor, so we write another helper method to do this.



Now we can write our publicly accessible method for adding Tasks to the database.



It turns out deleting from the database is a lot easier, so we won’t actually need any helper methods to write our second publicly accessible method.



Alternatively (maybe easier if you aren’t a SQL guru), you could write the body like this:



That’s it! Our database adapter is all set up.

Now create another instance variable, **mTaskAdapter**, in your **DisplayTasks** Activity and initialize it and open it during onCreate:

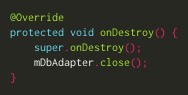
**private TaskAdapter mTaskAdapter;**

then within onCreate,

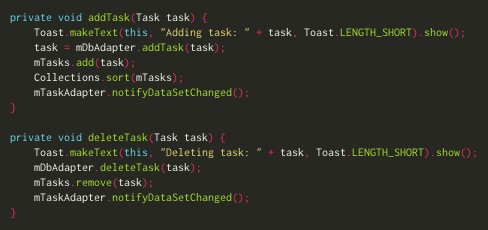
**mTaskAdapter = new TaskAdapter(this);**

**mTaskAdapter.open();**

We’ll also want to add an onDestroy method to handle closing the database connection when our application exits.



Finally, we fill in our **addTask** and **deleteTask** methods that we made at the beginning of this section.



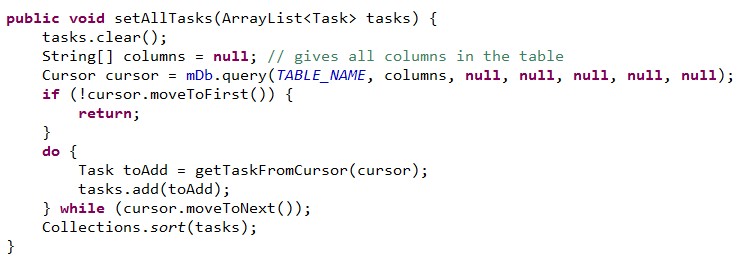
And we’re hooked up! Remove the code that inserts dummy values into the list. At this point you should test your application to make sure it’s working properly. You should be able to add Tasks from the menu option and delete them by long­pressing.

**Part E: Displaying Persistence**

Our database is currently storing all the information we want to store, but it isn’t loading and displaying it when we start up our app or when we rotate the screen. You have a couple options here.

First and easiest, you can continue to use an ArrayList. You will need to write a method that will populate the array list with the contents of the database, sorted so that tasks that are due first come first, and call it at the appropriate time or times. This will mean clearing the current tasks, querying the database for all entries, and iterating the cursor, making a Task for each table row (using your getTaskFromCursor() method).

Note that whenever you call setAllTasksAdapter, you will need to call mArrayAdapter.notifyDataSetChanged(); so the view reflects the changes to the list.



Second, which gives you more control over the appearance, you could do what we did in lecture: remove the array list, and implement getAllTasksCursor() in the database adapter, which returns a Cursor of all the tasks. To display the cursor, you will will also need to use a SimpleCursorAdapter instead of an ArrayAdapter, and call its changeCursor method whenever you add or remove a task. You’ll need to specify a custom layout of the row (in a new xml file), with a TextView for each field, like we did in lecture.