**Sqllite**

For sql installation refer this link:

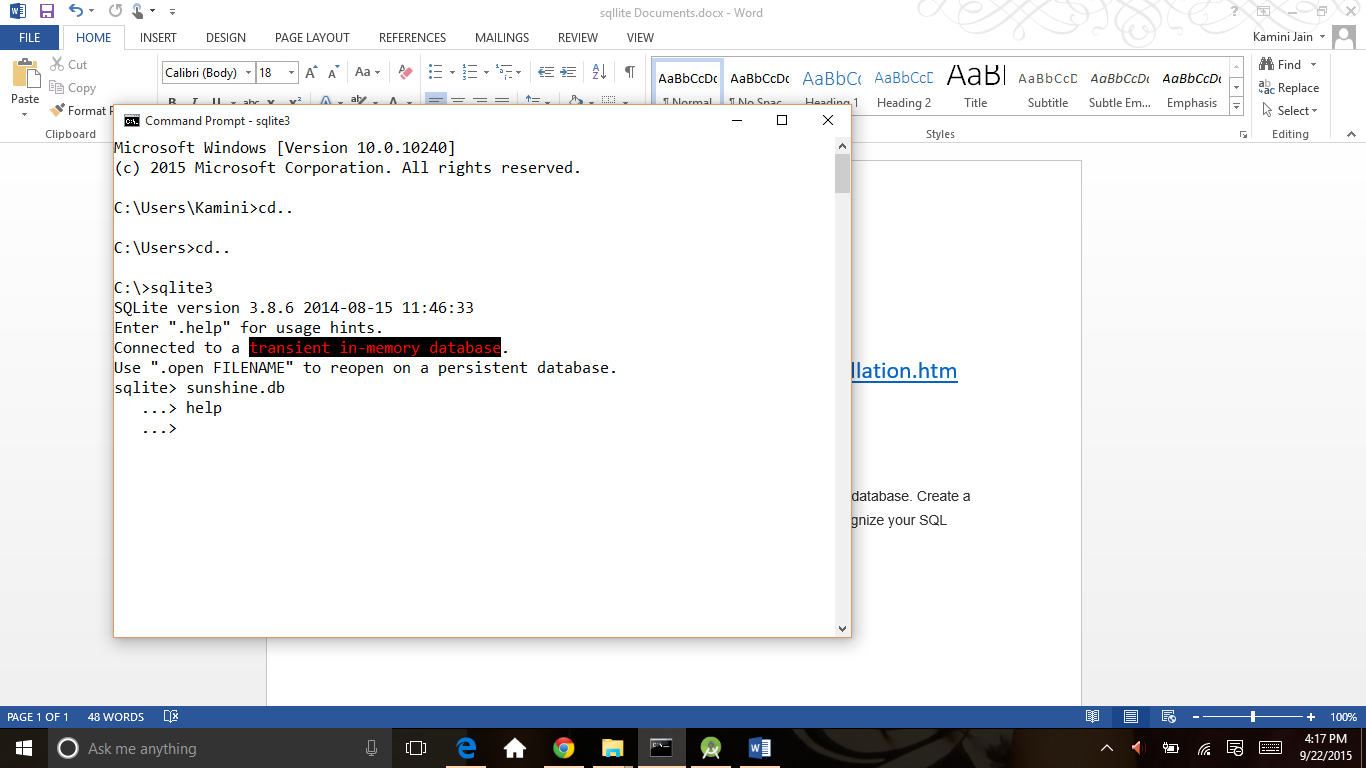
<http://www.tutorialspoint.com/sqlite/sqlite_installation.htm>

<http://www.sqlite.org/>

create database

Open terminal and navigate to a folder of your choice where you will save your database. Create a new database file called sunshine.db and start the SQLite shell (which will recognize your SQL commands) by typing:

sqlite3 sunshine.db



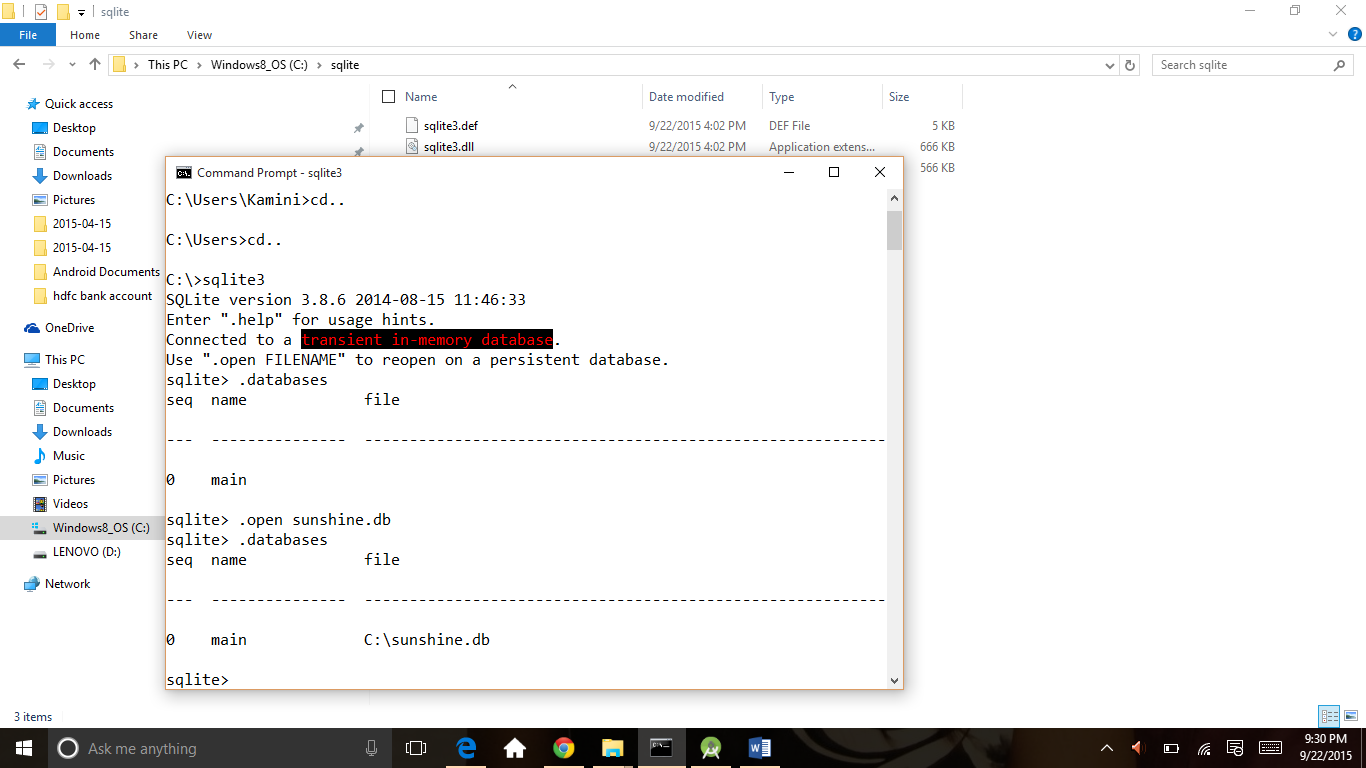
For a list of all commands:

.help

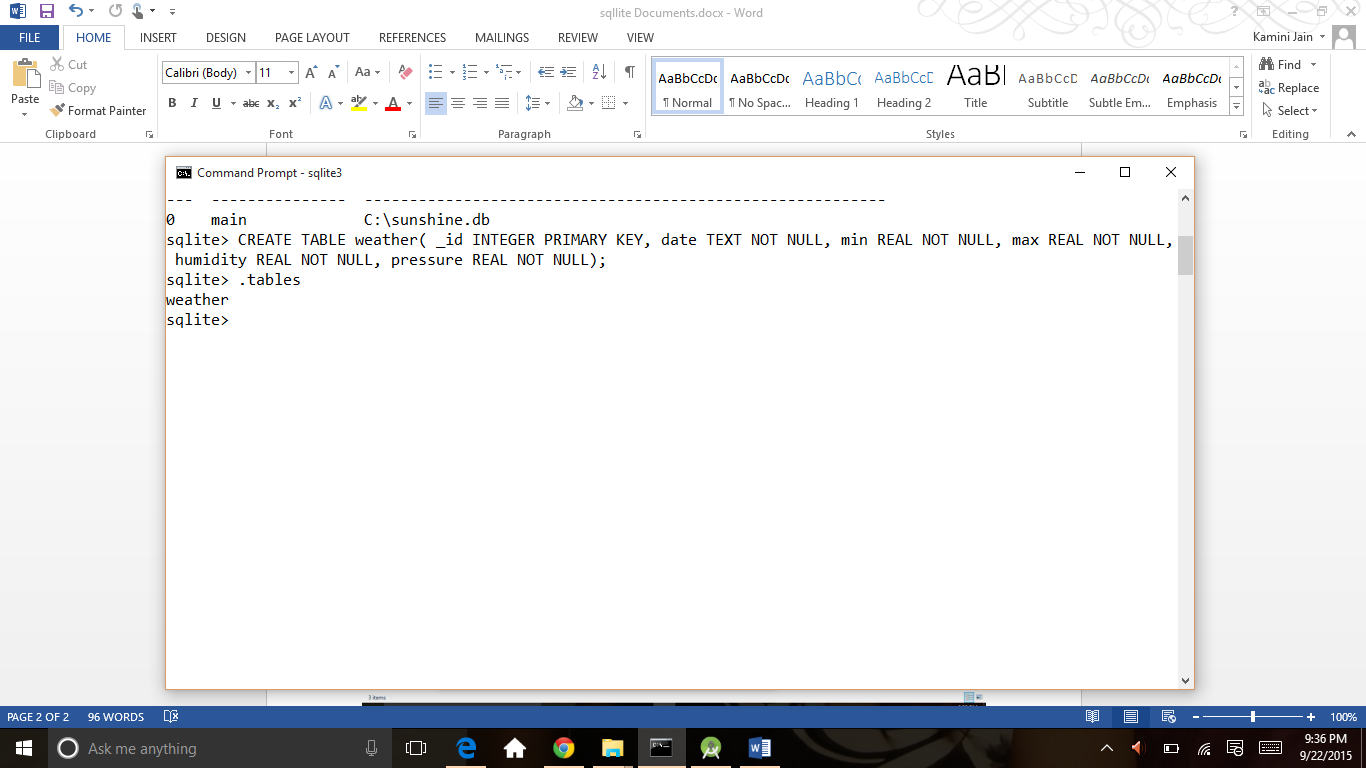
One of the commands is to list out all databases. It should display one database called main and the file location on your computer for sunshine.db.

.databases

First give sqllite>.open filename then use sqlite>.databases so it gives detalis see below screen

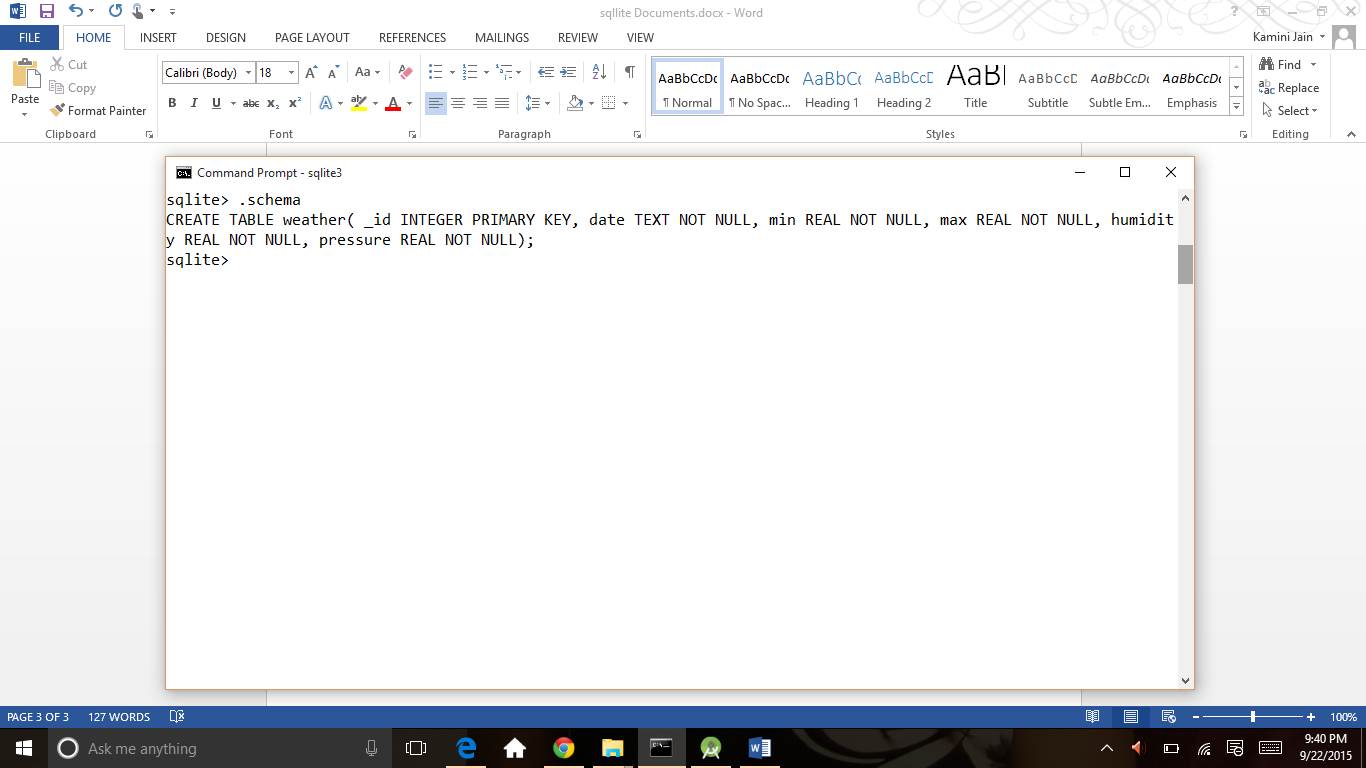


Create table using create command and then use .tables command to see tables in database



 you can find out the schema of how the tables were created in the database

.schema



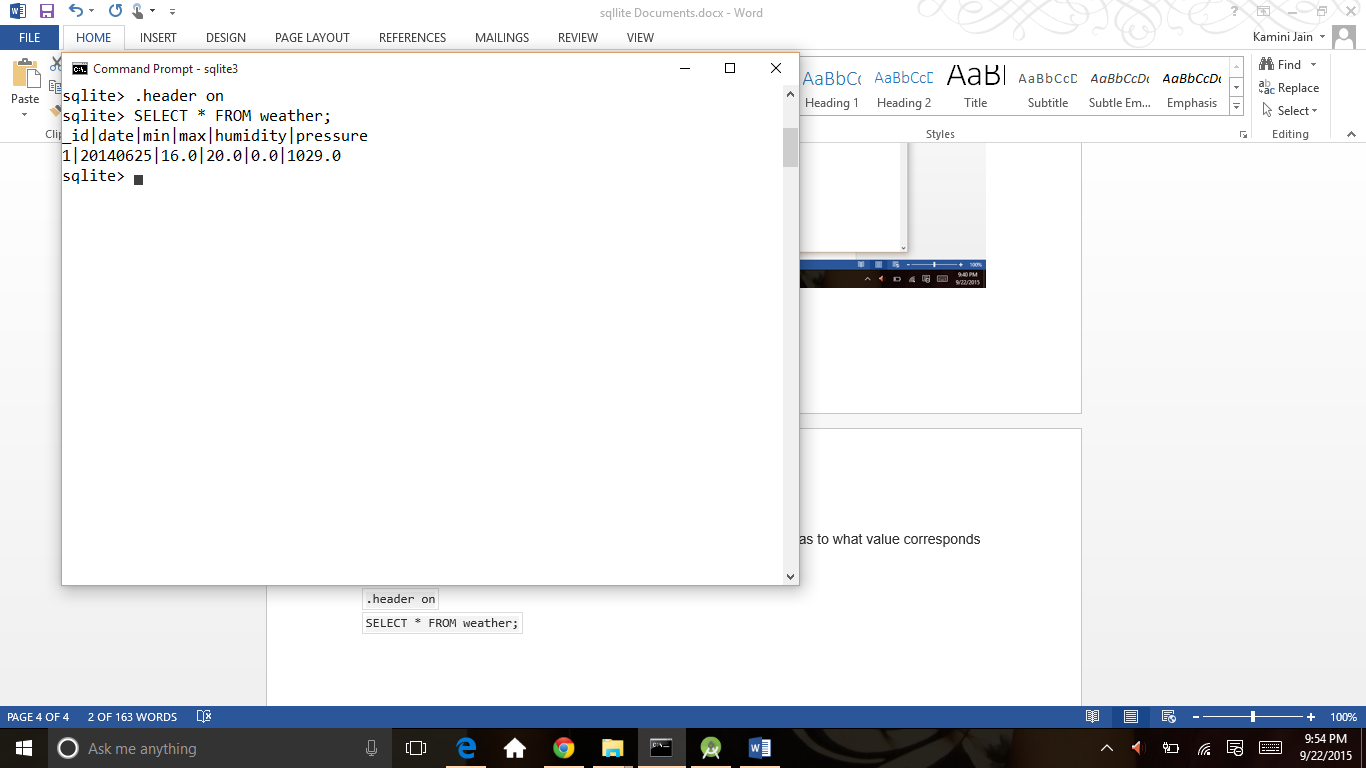
Insert values in table by insert query

INSERT INTO weather VALUES(1,'20140625',16,20,0,1029);

Always remember the semicolon at the end of a statement!

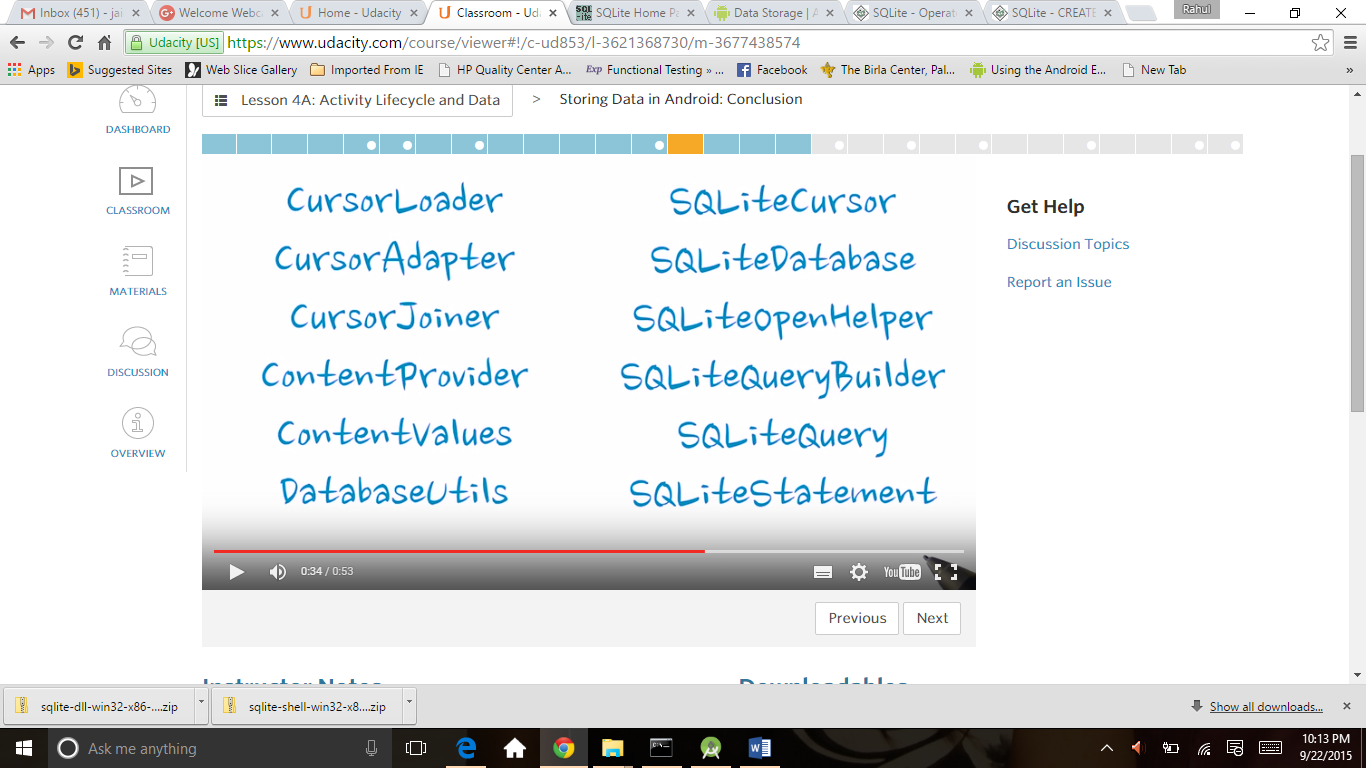
To have the column name be printed out as well (for easier readability as to what value corresponds to which column), turn the header on. Then do the query again.

.header on  
SELECT \* FROM weather;



For more about sqlite use above link:

<http://www.tutorialspoint.com/sqlite/sqlite_installation.htm>



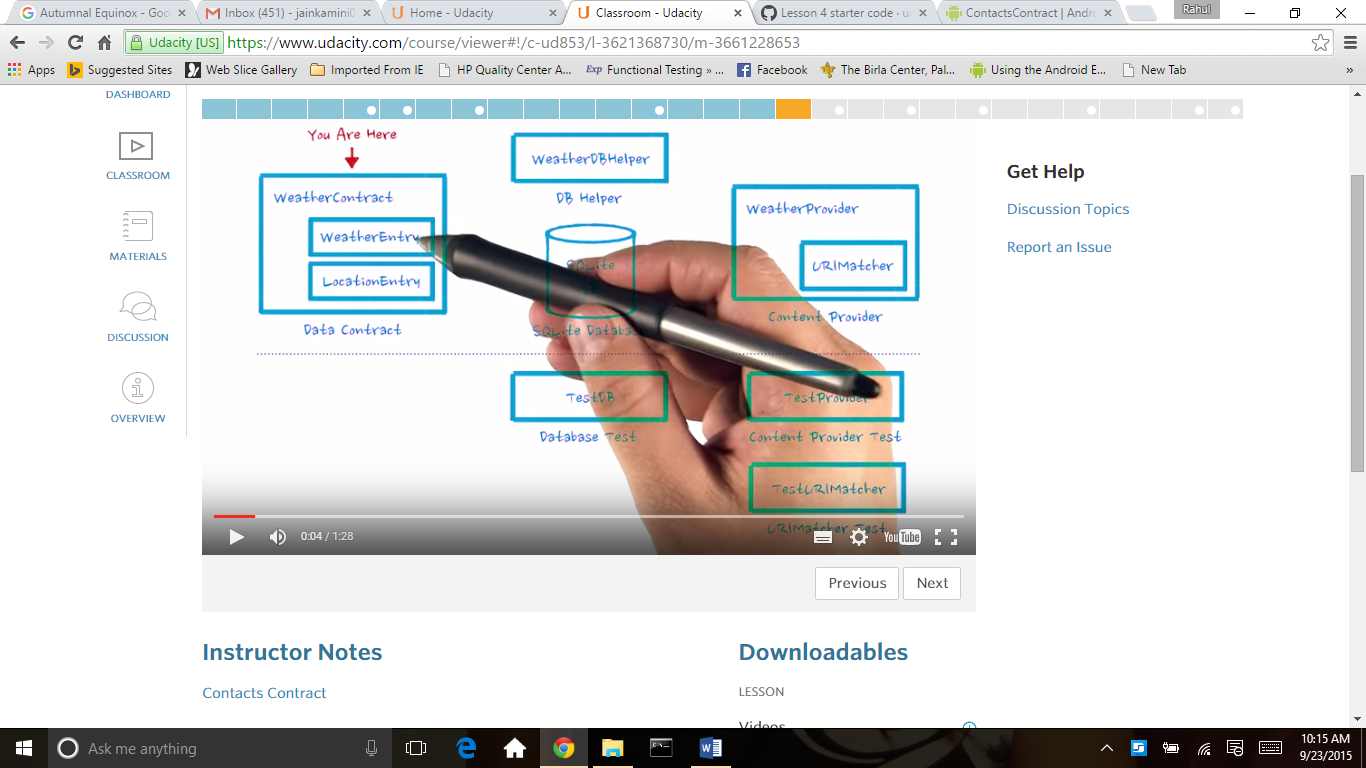
<http://developer.android.com/guide/topics/data/index.html>

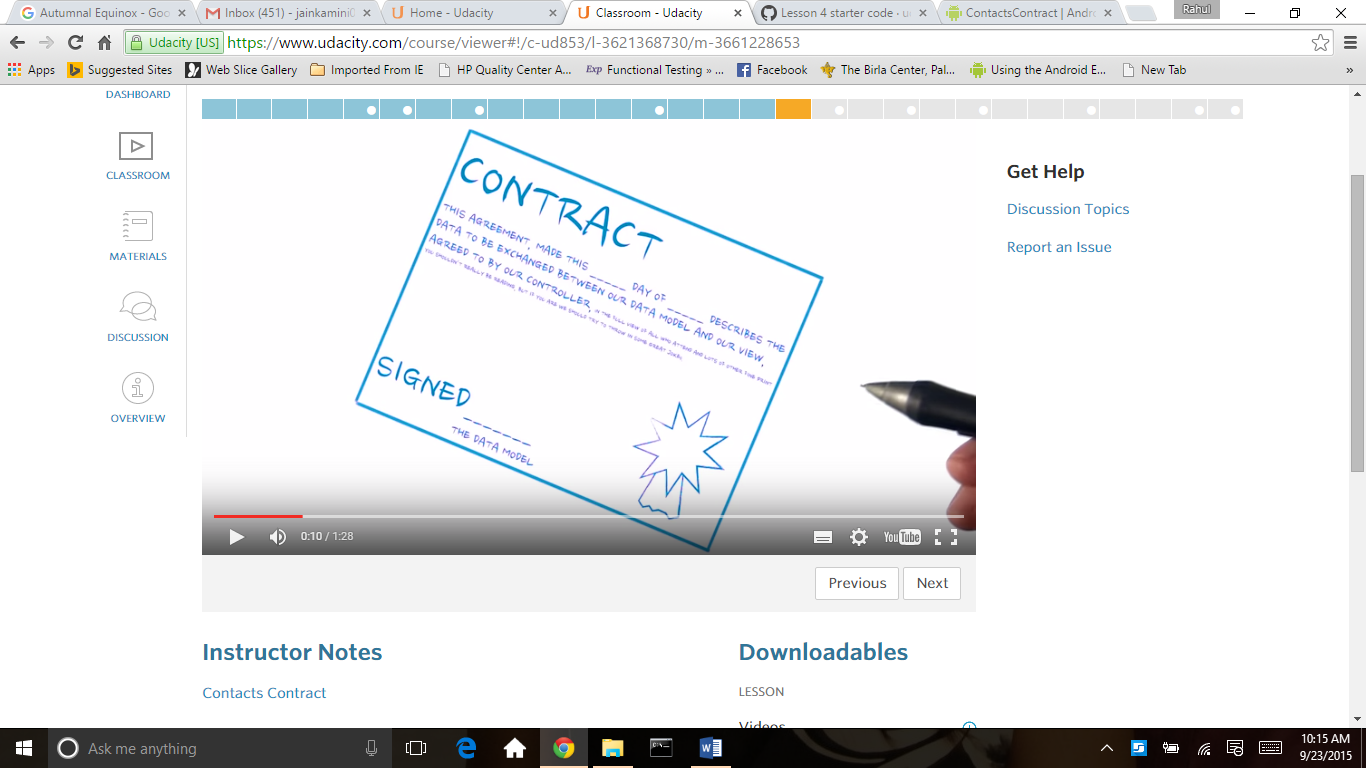
<http://developer.android.com/guide/topics/data/data-storage.html>

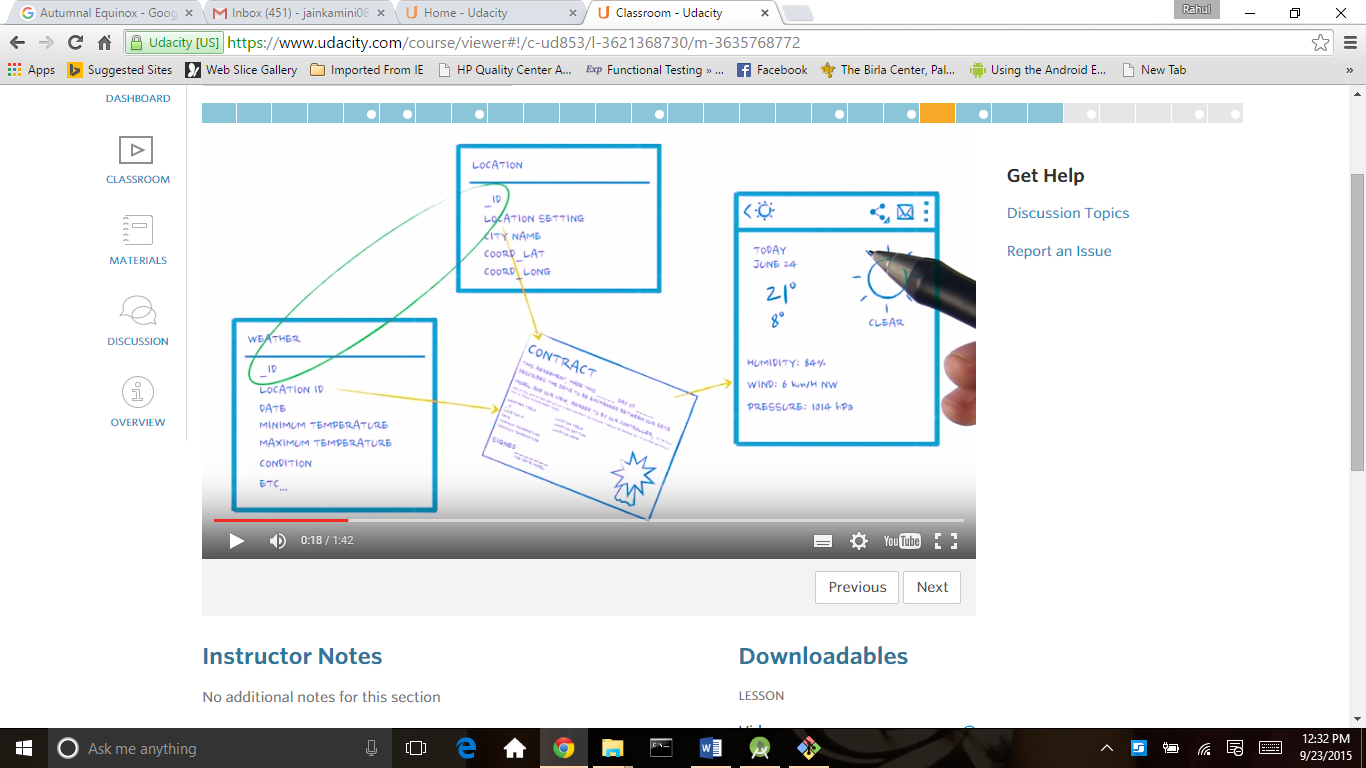
for contract

<http://developer.android.com/reference/android/provider/ContactsContract.html>

Contract is an agreement between data model, storage and views presentations describing how information is access.







Base column

<http://developer.android.com/reference/android/provider/BaseColumns.html>

content provider

<http://developer.android.com/guide/topics/providers/content-providers.html>

sqliteopen helper for create database .

<http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html>

<http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>

for app we use contract class for define all the calumns which is extend by Basecolumns class this is example of sunshine app WeatherContract class

*/\*  
 \* Copyright (C) 2014 The Android Open Source Project  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/***package** com.example.android.sunshine.app.data;  
  
**import** android.provider.BaseColumns;  
**import** android.text.format.Time;  
  
*/\*\*  
 \* Defines table and column names for the weather database.  
 \*/***public class** WeatherContract {  
  
 *// To make it easy to query for the exact date, we normalize all dates that go into  
 // the database to the start of the the Julian day at UTC.* **public static long** normalizeDate(**long** startDate) {  
 *// normalize the start date to the beginning of the (UTC) day* Time time = **new** Time();  
 time.set(startDate);  
 **int** julianDay = Time.*getJulianDay*(startDate, time.**gmtoff**);  
 **return** time.setJulianDay(julianDay);  
 }  
  
 */\*  
 Inner class that defines the table contents of the location table  
 Students: This is where you will add the strings. (Similar to what has been  
 done for WeatherEntry)  
 \*/* **public static final class** LocationEntry **implements** BaseColumns {  
 **public static final** String ***TABLE\_NAME*** = **"location"**;  
 *// The location setting string is what will be sent to openweathermap  
 // as the location query.* **public static final** String ***COLUMN\_LOCATION\_SETTING*** = **"location\_setting"**;  
  
 *// Human readable location string, provided by the API. Because for styling,  
 // "Mountain View" is more recognizable than 94043.* **public static final** String ***COLUMN\_CITY\_NAME*** = **"city\_name"**;  
  
 *// In order to uniquely pinpoint the location on the map when we launch the  
 // map intent, we store the latitude and longitude as returned by openweathermap.* **public static final** String ***COLUMN\_COORD\_LAT*** = **"coord\_lat"**;  
 **public static final** String ***COLUMN\_COORD\_LONG*** = **"coord\_long"**;  
  
 }  
  
 */\* Inner class that defines the table contents of the weather table \*/* **public static final class** WeatherEntry **implements** BaseColumns {  
  
 **public static final** String ***TABLE\_NAME*** = **"weather"**;  
  
 *// Column with the foreign key into the location table.* **public static final** String ***COLUMN\_LOC\_KEY*** = **"location\_id"**;  
 *// Date, stored as long in milliseconds since the epoch* **public static final** String ***COLUMN\_DATE*** = **"date"**;  
 *// Weather id as returned by API, to identify the icon to be used* **public static final** String ***COLUMN\_WEATHER\_ID*** = **"weather\_id"**;  
  
 *// Short description and long description of the weather, as provided by API.  
 // e.g "clear" vs "sky is clear".* **public static final** String ***COLUMN\_SHORT\_DESC*** = **"short\_desc"**;  
  
 *// Min and max temperatures for the day (stored as floats)* **public static final** String ***COLUMN\_MIN\_TEMP*** = **"min"**;  
 **public static final** String ***COLUMN\_MAX\_TEMP*** = **"max"**;  
  
 *// Humidity is stored as a float representing percentage* **public static final** String ***COLUMN\_HUMIDITY*** = **"humidity"**;  
  
 *// Humidity is stored as a float representing percentage* **public static final** String ***COLUMN\_PRESSURE*** = **"pressure"**;  
  
 *// Windspeed is stored as a float representing windspeed mph* **public static final** String ***COLUMN\_WIND\_SPEED*** = **"wind"**;  
  
 *// Degrees are meteorological degrees (e.g, 0 is north, 180 is south). Stored as floats.* **public static final** String ***COLUMN\_DEGREES*** = **"degrees"**;  
 }  
}

For database we use extend class of sqliteopenhelper

For sunshine we use WeatherDbHelper class for create database

*/\*  
 \* Copyright (C) 2014 The Android Open Source Project  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/***package** com.example.android.sunshine.app.data;  
  
**import** android.content.Context;  
**import** android.database.sqlite.SQLiteDatabase;  
**import** android.database.sqlite.SQLiteOpenHelper;  
  
**import** com.example.android.sunshine.app.data.WeatherContract.LocationEntry;  
**import** com.example.android.sunshine.app.data.WeatherContract.WeatherEntry;  
  
*/\*\*  
 \* Manages a local database for weather data.  
 \*/***public class** WeatherDbHelper **extends** SQLiteOpenHelper {  
  
 *// If you change the database schema, you must increment the database version.* **private static final int *DATABASE\_VERSION*** = 2;  
  
 **static final** String ***DATABASE\_NAME*** = **"weather.db"**;  
  
 **public** WeatherDbHelper(Context context) {  
 **super**(context, ***DATABASE\_NAME***, **null**, ***DATABASE\_VERSION***);  
 }  
  
 @Override  
 **public void** onCreate(SQLiteDatabase sqLiteDatabase) {  
 *// Create a table to hold locations. A location consists of the string supplied in the  
 // location setting, the city name, and the latitude and longitude* **final** String SQL\_CREATE\_LOCATION\_TABLE = **"CREATE TABLE "** + LocationEntry.***TABLE\_NAME*** + **" ("** +  
 LocationEntry.***\_ID*** + **" INTEGER PRIMARY KEY,"** +  
 LocationEntry.***COLUMN\_LOCATION\_SETTING*** + **" TEXT UNIQUE NOT NULL, "** +  
 LocationEntry.***COLUMN\_CITY\_NAME*** + **" TEXT NOT NULL, "** +  
 LocationEntry.***COLUMN\_COORD\_LAT*** + **" REAL NOT NULL, "** +  
 LocationEntry.***COLUMN\_COORD\_LONG*** + **" REAL NOT NULL "** +  
 **" );"**;  
  
 **final** String SQL\_CREATE\_WEATHER\_TABLE = **"CREATE TABLE "** + WeatherEntry.***TABLE\_NAME*** + **" ("** +  
 *// Why AutoIncrement here, and not above?  
 // Unique keys will be auto-generated in either case. But for weather  
 // forecasting, it's reasonable to assume the user will want information  
 // for a certain date and all dates \*following\*, so the forecast data  
 // should be sorted accordingly.* WeatherEntry.***\_ID*** + **" INTEGER PRIMARY KEY AUTOINCREMENT,"** +  
  
 *// the ID of the location entry associated with this weather data* WeatherEntry.***COLUMN\_LOC\_KEY*** + **" INTEGER NOT NULL, "** +  
 WeatherEntry.***COLUMN\_DATE*** + **" INTEGER NOT NULL, "** +  
 WeatherEntry.***COLUMN\_SHORT\_DESC*** + **" TEXT NOT NULL, "** +  
 WeatherEntry.***COLUMN\_WEATHER\_ID*** + **" INTEGER NOT NULL,"** +  
  
 WeatherEntry.***COLUMN\_MIN\_TEMP*** + **" REAL NOT NULL, "** +  
 WeatherEntry.***COLUMN\_MAX\_TEMP*** + **" REAL NOT NULL, "** +  
  
 WeatherEntry.***COLUMN\_HUMIDITY*** + **" REAL NOT NULL, "** +  
 WeatherEntry.***COLUMN\_PRESSURE*** + **" REAL NOT NULL, "** +  
 WeatherEntry.***COLUMN\_WIND\_SPEED*** + **" REAL NOT NULL, "** +  
 WeatherEntry.***COLUMN\_DEGREES*** + **" REAL NOT NULL, "** +  
  
 *// Set up the location column as a foreign key to location table.* **" FOREIGN KEY ("** + WeatherEntry.***COLUMN\_LOC\_KEY*** + **") REFERENCES "** +  
 LocationEntry.***TABLE\_NAME*** + **" ("** + LocationEntry.***\_ID*** + **"), "** +  
  
 *// To assure the application have just one weather entry per day  
 // per location, it's created a UNIQUE constraint with REPLACE strategy* **" UNIQUE ("** + WeatherEntry.***COLUMN\_DATE*** + **", "** +  
 WeatherEntry.***COLUMN\_LOC\_KEY*** + **") ON CONFLICT REPLACE);"**;  
  
 sqLiteDatabase.execSQL(SQL\_CREATE\_LOCATION\_TABLE);  
 sqLiteDatabase.execSQL(SQL\_CREATE\_WEATHER\_TABLE);  
 }  
  
 @Override  
 **public void** onUpgrade(SQLiteDatabase sqLiteDatabase, **int** oldVersion, **int** newVersion) {  
 *// This database is only a cache for online data, so its upgrade policy is  
 // to simply to discard the data and start over  
 // Note that this only fires if you change the version number for your database.  
 // It does NOT depend on the version number for your application.  
 // If you want to update the schema without wiping data, commenting out the next 2 lines  
 // should be your top priority before modifying this method.* sqLiteDatabase.execSQL(**"DROP TABLE IF EXISTS "** + LocationEntry.***TABLE\_NAME***);  
 sqLiteDatabase.execSQL(**"DROP TABLE IF EXISTS "** + WeatherEntry.***TABLE\_NAME***);  
 onCreate(sqLiteDatabase);  
 }  
}

Testing fundamentals

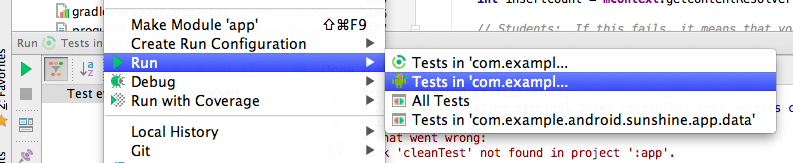
<http://developer.android.com/tools/testing/testing_android.html#TestResults>

<http://www.tutorialspoint.com/junit/junit_quick_guide.htm>

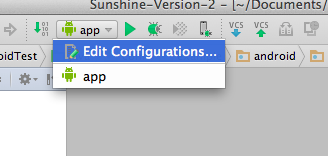
There are two ways to run tests.

Option1

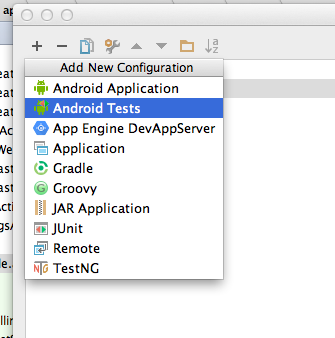
Right click on the folder **androidTest** and select **Run** > **Tests in ‘com.exampl…**. Make sure to select the Android logo (as shown below) and **not** the Gradle logo.



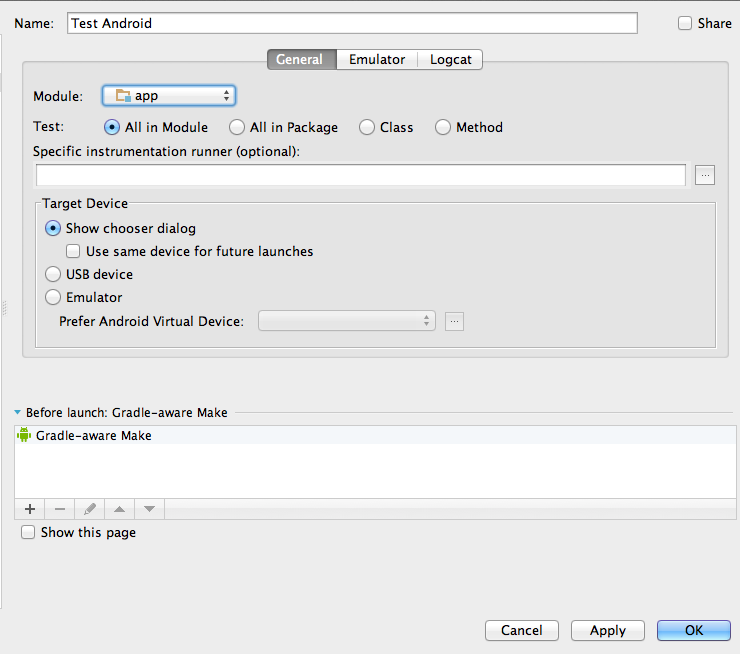
Option 2 :Right next to the **Run** button, click on the drop down and select **Edit Configurations**.



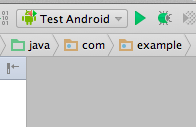
You’re going to make a “testing” configuration. To do this, click the **+** button and select **Android Tests**.



Make sure the module is **app** and name it something related, such as “Test Android”.



Then you can hit **OK**. Finally, press the run button, with your test configuration selected in the drop down, to run the tests.



If you see any error messages when you try to run your tests, make sure there aren't any **JUnit** run configurations (we only want **Android Tests**). If there's anything under the **JUnit** dropdown in the left section, try deleting it (select it and click the **-** button).

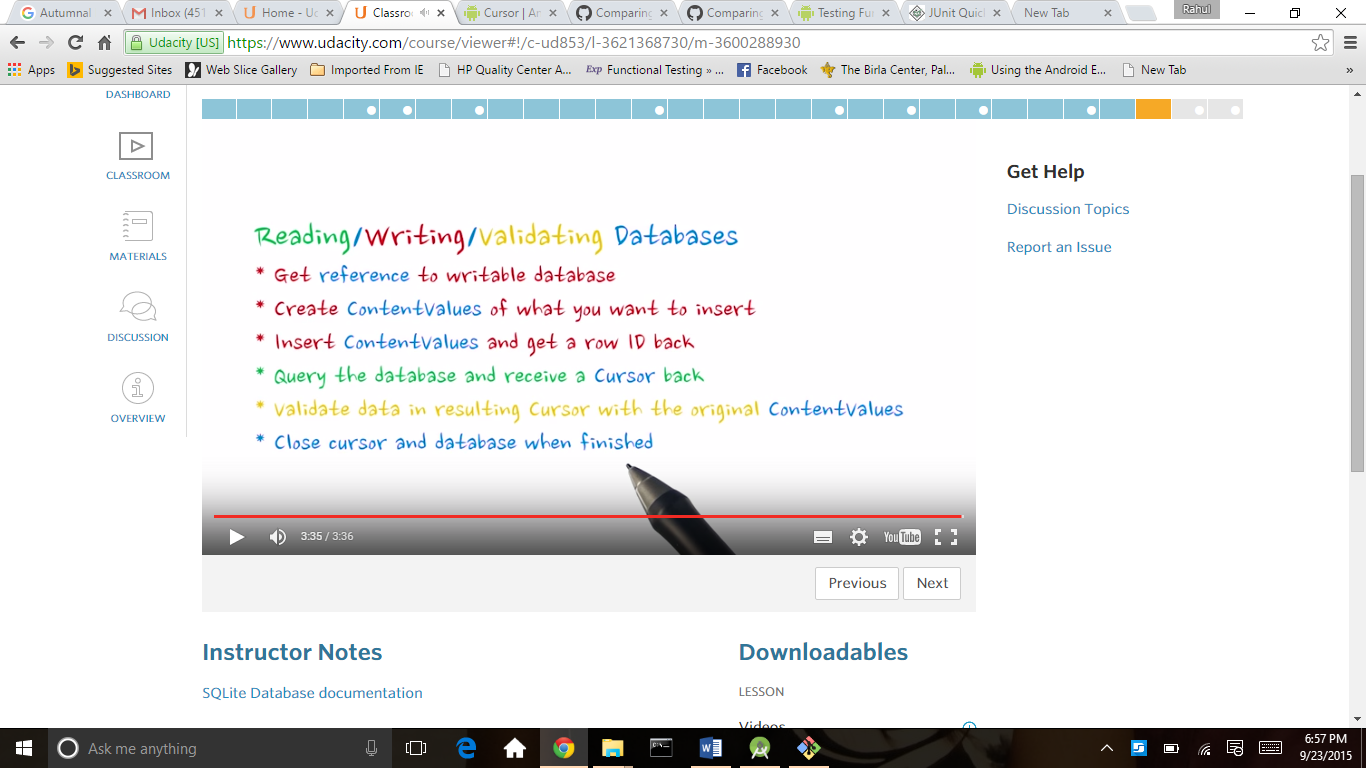
This is for use for testing purpose like reading /writing /validate database

Code of below is Testdb class

In this class we testCreateDb() function for test create database and add tables in it .

testWeatherTable() method is use for test select functions in table like below given screen

TestUtilities class : this class is extend from AndroidTestCase in this test insertvalue in table and return select value from .



*/\*  
 \* Copyright (C) 2014 The Android Open Source Project  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/***package** com.example.android.sunshine.app.data;  
  
**import** android.content.ContentValues;  
**import** android.database.Cursor;  
**import** android.database.sqlite.SQLiteDatabase;  
**import** android.test.AndroidTestCase;  
  
**import** java.util.HashSet;  
  
**public class** TestDb **extends** AndroidTestCase {  
  
 **public static final** String ***LOG\_TAG*** = TestDb.**class**.getSimpleName();  
  
 *// Since we want each test to start with a clean slate* **void** deleteTheDatabase() {  
 **mContext**.deleteDatabase(WeatherDbHelper.***DATABASE\_NAME***);  
 }  
  
 */\*  
 This function gets called before each test is executed to delete the database. This makes  
 sure that we always have a clean test.  
 \*/* **public void** setUp() {  
 deleteTheDatabase();  
 }  
  
 */\*  
 Students: Uncomment this test once you've written the code to create the Location  
 table. Note that you will have to have chosen the same column names that I did in  
 my solution for this test to compile, so if you haven't yet done that, this is  
 a good time to change your column names to match mine.  
  
 Note that this only tests that the Location table has the correct columns, since we  
 give you the code for the weather table. This test does not look at the  
 \*/* **public void** testCreateDb() **throws** Throwable {  
 *// build a HashSet of all of the table names we wish to look for  
 // Note that there will be another table in the DB that stores the  
 // Android metadata (db version information)* **final** HashSet<String> tableNameHashSet = **new** HashSet<String>();  
 tableNameHashSet.add(WeatherContract.LocationEntry.***TABLE\_NAME***);  
 tableNameHashSet.add(WeatherContract.WeatherEntry.***TABLE\_NAME***);  
  
 **mContext**.deleteDatabase(WeatherDbHelper.***DATABASE\_NAME***);  
 SQLiteDatabase db = **new** WeatherDbHelper(  
 **this**.**mContext**).getWritableDatabase();  
 *assertEquals*(**true**, db.isOpen());  
  
 *// have we created the tables we want?* Cursor c = db.rawQuery(**"SELECT name FROM sqlite\_master WHERE type='table'"**, **null**);  
  
 *assertTrue*(**"Error: This means that the database has not been created correctly"**,  
 c.moveToFirst());  
  
 *// verify that the tables have been created* **do** {  
 tableNameHashSet.remove(c.getString(0));  
 } **while**( c.moveToNext() );  
  
 *// if this fails, it means that your database doesn't contain both the location entry  
 // and weather entry tables  
 assertTrue*(**"Error: Your database was created without both the location entry and weather entry tables"**,  
 tableNameHashSet.isEmpty());  
  
 *// now, do our tables contain the correct columns?* c = db.rawQuery(**"PRAGMA table\_info("** + WeatherContract.LocationEntry.***TABLE\_NAME*** + **")"**,  
 **null**);  
  
 *assertTrue*(**"Error: This means that we were unable to query the database for table information."**,  
 c.moveToFirst());  
  
 *// Build a HashSet of all of the column names we want to look for* **final** HashSet<String> locationColumnHashSet = **new** HashSet<String>();  
 locationColumnHashSet.add(WeatherContract.LocationEntry.***\_ID***);  
 locationColumnHashSet.add(WeatherContract.LocationEntry.***COLUMN\_CITY\_NAME***);  
 locationColumnHashSet.add(WeatherContract.LocationEntry.***COLUMN\_COORD\_LAT***);  
 locationColumnHashSet.add(WeatherContract.LocationEntry.***COLUMN\_COORD\_LONG***);  
 locationColumnHashSet.add(WeatherContract.LocationEntry.***COLUMN\_LOCATION\_SETTING***);  
  
 **int** columnNameIndex = c.getColumnIndex(**"name"**);  
 **do** {  
 String columnName = c.getString(columnNameIndex);  
 locationColumnHashSet.remove(columnName);  
 } **while**(c.moveToNext());  
  
 *// if this fails, it means that your database doesn't contain all of the required location  
 // entry columns  
 assertTrue*(**"Error: The database doesn't contain all of the required location entry columns"**,  
 locationColumnHashSet.isEmpty());  
 db.close();  
 }  
  
 */\*  
 Students: Here is where you will build code to test that we can insert and query the  
 location database. We've done a lot of work for you. You'll want to look in TestUtilities  
 where you can uncomment out the "createNorthPoleLocationValues" function. You can  
 also make use of the ValidateCurrentRecord function from within TestUtilities.  
 \*/* **public void** testLocationTable() {  
 insertLocation();  
 }  
  
 */\*  
 Students: Here is where you will build code to test that we can insert and query the  
 database. We've done a lot of work for you. You'll want to look in TestUtilities  
 where you can use the "createWeatherValues" function. You can  
 also make use of the validateCurrentRecord function from within TestUtilities.  
 \*/* **public void** testWeatherTable() {  
 *// First insert the location, and then use the locationRowId to insert  
 // the weather. Make sure to cover as many failure cases as you can.  
  
 // Instead of rewriting all of the code we've already written in testLocationTable  
 // we can move this code to insertLocation and then call insertLocation from both  
 // tests. Why move it? We need the code to return the ID of the inserted location  
 // and our testLocationTable can only return void because it's a test.* **long** locationRowId = insertLocation();  
  
 *// Make sure we have a valid row ID.  
 assertFalse*(**"Error: Location Not Inserted Correctly"**, locationRowId == -1L);  
  
 *// First step: Get reference to writable database  
 // If there's an error in those massive SQL table creation Strings,  
 // errors will be thrown here when you try to get a writable database.* WeatherDbHelper dbHelper = **new** WeatherDbHelper(**mContext**);  
 SQLiteDatabase db = dbHelper.getWritableDatabase();  
  
 *// Second Step (Weather): Create weather values* ContentValues weatherValues = TestUtilities.*createWeatherValues*(locationRowId);  
  
 *// Third Step (Weather): Insert ContentValues into database and get a row ID back* **long** weatherRowId = db.insert(WeatherContract.WeatherEntry.***TABLE\_NAME***, **null**, weatherValues);  
 *assertTrue*(weatherRowId != -1);  
  
 *// Fourth Step: Query the database and receive a Cursor back  
 // A cursor is your primary interface to the query results.* Cursor weatherCursor = db.query(  
 WeatherContract.WeatherEntry.***TABLE\_NAME***, *// Table to Query* **null**, *// leaving "columns" null just returns all the columns.* **null**, *// cols for "where" clause* **null**, *// values for "where" clause* **null**, *// columns to group by* **null**, *// columns to filter by row groups* **null** *// sort order* );  
  
 *// Move the cursor to the first valid database row and check to see if we have any rows  
 assertTrue*( **"Error: No Records returned from location query"**, weatherCursor.moveToFirst() );  
  
 *// Fifth Step: Validate the location Query* TestUtilities.*validateCurrentRecord*(**"testInsertReadDb weatherEntry failed to validate"**,  
 weatherCursor, weatherValues);  
  
 *// Move the cursor to demonstrate that there is only one record in the database  
 assertFalse*( **"Error: More than one record returned from weather query"**,  
 weatherCursor.moveToNext() );  
  
 *// Sixth Step: Close cursor and database* weatherCursor.close();  
 dbHelper.close();  
 }  
  
  
 */\*  
 Students: This is a helper method for the testWeatherTable quiz. You can move your  
 code from testLocationTable to here so that you can call this code from both  
 testWeatherTable and testLocationTable.  
 \*/* **public long** insertLocation() {  
 *// First step: Get reference to writable database  
 // If there's an error in those massive SQL table creation Strings,  
 // errors will be thrown here when you try to get a writable database.* WeatherDbHelper dbHelper = **new** WeatherDbHelper(**mContext**);  
 SQLiteDatabase db = dbHelper.getWritableDatabase();  
  
 *// Second Step: Create ContentValues of what you want to insert  
 // (you can use the createNorthPoleLocationValues if you wish)* ContentValues testValues = TestUtilities.*createNorthPoleLocationValues*();  
  
 *// Third Step: Insert ContentValues into database and get a row ID back* **long** locationRowId;  
 locationRowId = db.insert(WeatherContract.LocationEntry.***TABLE\_NAME***, **null**, testValues);  
  
 *// Verify we got a row back.  
 assertTrue*(locationRowId != -1);  
  
 *// Data's inserted. IN THEORY. Now pull some out to stare at it and verify it made  
 // the round trip.  
  
 // Fourth Step: Query the database and receive a Cursor back  
 // A cursor is your primary interface to the query results.* Cursor cursor = db.query(  
 WeatherContract.LocationEntry.***TABLE\_NAME***, *// Table to Query* **null**, *// all columns* **null**, *// Columns for the "where" clause* **null**, *// Values for the "where" clause* **null**, *// columns to group by* **null**, *// columns to filter by row groups* **null** *// sort order* );  
  
 *// Move the cursor to a valid database row and check to see if we got any records back  
 // from the query  
 assertTrue*( **"Error: No Records returned from location query"**, cursor.moveToFirst() );  
  
 *// Fifth Step: Validate data in resulting Cursor with the original ContentValues  
 // (you can use the validateCurrentRecord function in TestUtilities to validate the  
 // query if you like)* TestUtilities.*validateCurrentRecord*(**"Error: Location Query Validation Failed"**,  
 cursor, testValues);  
  
 *// Move the cursor to demonstrate that there is only one record in the database  
 assertFalse*( **"Error: More than one record returned from location query"**,  
 cursor.moveToNext() );  
  
 *// Sixth Step: Close Cursor and Database* cursor.close();  
 db.close();  
 **return** locationRowId;  
 }  
}

**package** com.example.android.sunshine.app.data;  
  
**import** android.content.ContentValues;  
**import** android.content.Context;  
**import** android.database.ContentObserver;  
**import** android.database.Cursor;  
**import** android.database.sqlite.SQLiteDatabase;  
**import** android.net.Uri;  
**import** android.os.Handler;  
**import** android.os.HandlerThread;  
**import** android.test.AndroidTestCase;  
  
**import** com.example.android.sunshine.app.utils.PollingCheck;  
  
**import** java.util.Map;  
**import** java.util.Set;  
  
*/\*  
 Students: These are functions and some test data to make it easier to test your database and  
 Content Provider. Note that you'll want your WeatherContract class to exactly match the one  
 in our solution to use these as-given.  
 \*/***public class** TestUtilities **extends** AndroidTestCase {  
 **static final** String ***TEST\_LOCATION*** = **"99705"**;  
 **static final long *TEST\_DATE*** = 1419033600L; *// December 20th, 2014* **static void** validateCursor(String error, Cursor valueCursor, ContentValues expectedValues) {  
 *assertTrue*(**"Empty cursor returned. "** + error, valueCursor.moveToFirst());  
 *validateCurrentRecord*(error, valueCursor, expectedValues);  
 valueCursor.close();  
 }  
  
 **static void** validateCurrentRecord(String error, Cursor valueCursor, ContentValues expectedValues) {  
 Set<Map.Entry<String, Object>> valueSet = expectedValues.valueSet();  
 **for** (Map.Entry<String, Object> entry : valueSet) {  
 String columnName = entry.getKey();  
 **int** idx = valueCursor.getColumnIndex(columnName);  
 *assertFalse*(**"Column '"** + columnName + **"' not found. "** + error, idx == -1);  
 String expectedValue = entry.getValue().toString();  
 *assertEquals*(**"Value '"** + entry.getValue().toString() +  
 **"' did not match the expected value '"** +  
 expectedValue + **"'. "** + error, expectedValue, valueCursor.getString(idx));  
 }  
 }  
  
 */\*  
 Students: Use this to create some default weather values for your database tests.  
 \*/* **static** ContentValues createWeatherValues(**long** locationRowId) {  
 ContentValues weatherValues = **new** ContentValues();  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_LOC\_KEY***, locationRowId);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_DATE***, ***TEST\_DATE***);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_DEGREES***, 1.1);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_HUMIDITY***, 1.2);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_PRESSURE***, 1.3);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_MAX\_TEMP***, 75);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_MIN\_TEMP***, 65);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_SHORT\_DESC***, **"Asteroids"**);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_WIND\_SPEED***, 5.5);  
 weatherValues.put(WeatherContract.WeatherEntry.***COLUMN\_WEATHER\_ID***, 321);  
  
 **return** weatherValues;  
 }  
  
 */\*  
 Students: You can uncomment this helper function once you have finished creating the  
 LocationEntry part of the WeatherContract.  
 \*/* **static** ContentValues createNorthPoleLocationValues() {  
 *// Create a new map of values, where column names are the keys* ContentValues testValues = **new** ContentValues();  
 testValues.put(WeatherContract.LocationEntry.***COLUMN\_LOCATION\_SETTING***, ***TEST\_LOCATION***);  
 testValues.put(WeatherContract.LocationEntry.***COLUMN\_CITY\_NAME***, **"North Pole"**);  
 testValues.put(WeatherContract.LocationEntry.***COLUMN\_COORD\_LAT***, 64.7488);  
 testValues.put(WeatherContract.LocationEntry.***COLUMN\_COORD\_LONG***, -147.353);  
  
 **return** testValues;  
 }  
  
 */\*  
 Students: You can uncomment this function once you have finished creating the  
 LocationEntry part of the WeatherContract as well as the WeatherDbHelper.  
 \*/* **static long** insertNorthPoleLocationValues(Context context) {  
 *// insert our test records into the database* WeatherDbHelper dbHelper = **new** WeatherDbHelper(context);  
 SQLiteDatabase db = dbHelper.getWritableDatabase();  
 ContentValues testValues = TestUtilities.*createNorthPoleLocationValues*();  
  
 **long** locationRowId;  
 locationRowId = db.insert(WeatherContract.LocationEntry.***TABLE\_NAME***, **null**, testValues);  
  
 *// Verify we got a row back.  
 assertTrue*(**"Error: Failure to insert North Pole Location Values"**, locationRowId != -1);  
  
 **return** locationRowId;  
 }  
  
 */\*  
 Students: The functions we provide inside of TestProvider use this utility class to test  
 the ContentObserver callbacks using the PollingCheck class that we grabbed from the Android  
 CTS tests.  
  
 Note that this only tests that the onChange function is called; it does not test that the  
 correct Uri is returned.  
 \*/* **static class** TestContentObserver **extends** ContentObserver {  
 **final** HandlerThread **mHT**;  
 **boolean mContentChanged**;  
  
 **static** TestContentObserver getTestContentObserver() {  
 HandlerThread ht = **new** HandlerThread(**"ContentObserverThread"**);  
 ht.start();  
 **return new** TestContentObserver(ht);  
 }  
  
 **private** TestContentObserver(HandlerThread ht) {  
 **super**(**new** Handler(ht.getLooper()));  
 **mHT** = ht;  
 }  
  
 *// On earlier versions of Android, this onChange method is called* @Override  
 **public void** onChange(**boolean** selfChange) {  
 onChange(selfChange, **null**);  
 }  
  
 @Override  
 **public void** onChange(**boolean** selfChange, Uri uri) {  
 **mContentChanged** = **true**;  
 }  
  
 **public void** waitForNotificationOrFail() {  
 *// Note: The PollingCheck class is taken from the Android CTS (Compatibility Test Suite).  
 // It's useful to look at the Android CTS source for ideas on how to test your Android  
 // applications. The reason that PollingCheck works is that, by default, the JUnit  
 // testing framework is not running on the main Android application thread.* **new** PollingCheck(5000) {  
 @Override  
 **protected boolean** check() {  
 **return mContentChanged**;  
 }  
 }.run();  
 **mHT**.quit();  
 }  
 }  
  
 **static** TestContentObserver getTestContentObserver() {  
 **return** TestContentObserver.*getTestContentObserver*();  
 }  
}