Lab 4: Joke List 3.0

Intro

This lab will be a continuation of Lab 3. You will be given a solution to Lab 3 so you may begin Lab 4 even if you did not complete all of Lab 3. You will be persisting data using a content provider with a SQLite database inside, and preserving the state of an application during its life cycle. It is important to note that this lab is meant to be done in order, from start to finish. Each activity builds on the previous one, so skipping over earlier activities in the lab may cause you to miss an important lesson that you should be using in later activities.

Objectives

At the end of this lab you will be expected to know:

- *How to save & restore data as Application Preferences.*
- *How to save & restore data as Instance State.*
- *How to create, maintain and interact with Content Providers and tables.*
    - *How to implement SQLite database helper classes to aid in table access, database initialization and minimal management.*
    - *How to access and modify a SQLite database table using URIs and content provider database calls.*

- *What Cursors and CursorAdapters are and how they interact with Views.*
    - *How to asynchronously load and auto-manage Cursors using LoaderManager and CursorLoader.*

Activities

For this lab we will be extending the "Joke List" application that you worked on in Lab 3. The previous app you created functioned completely on its own, but when the orientation or state of the device was changed the jokes disappeared, replaced by the original unrated prepopulated jokes. This version of the app will add a persistence layer to the previous version to fix these issues and more. It will allow Jokes that are added to be saved to a database as well as maintain application state throughout the life cycle of the application. All tasks for this lab will be based off of this application. Over the course of the lab you will be iteratively refining and adding functionality to the Joke List app. With each iteration you will be improving upon the previous iteration's functionality.

IMPORTANT:
You will be given a Skeleton Project to work with. This project contains all of the java and resource files you will need to complete the lab. Some method stubs, member variables, and resource values and ids have been added as well. It is important that you not change the names of these methods, variables, and resource values and ids. These are given to you because there are unit tests included in this project that depend on these items being declared exactly as they are. These units test will be used to evaluate the correctness of your lab. You have complete access to these test cases during development, which gives you the ability to run these tests yourself. In fact, you are encouraged to run these tests to ensure that your application is functioning properly.

Contents

1. Setting Up

1.1 Setting Up the Project

To begin, download and extract the skeleton project for the JokeList application from Polylearn.

- Extract the project, making sure to preserve the folder structure.
    - *Take note of the path to the root folder of the skeleton project. You may prefer to extract it to your Eclipse workspace directory.*
- ***Note: If you have a fully functional lab 3 implementation, you can use that as a basis and add new methods, classes, features and resources found in the skeleton project to it, or vice-versa.*** *Best to use a code base you're familiar with, provided it works as intended.*

Next you will need to set up the Joke List Android project for this app. Since the skeleton project was created in Eclipse, the easiest thing is to import this project into Eclipse.

- Select **File -> Import...**.
- In the Import Wizard, expand **General** and select **Existing Projects into Workspace**. Click **Next**.
- In the Import Project wizard, click **Select root directory** and click Browse. Select the root directory of the skeleton project that you extracted. Click Open and then Finish.
- Click on the project name in the Package Explorer. Select File -> Rename and change the name of your project to **lab4<userid>** where <userid> is your user id (e.g. jsmith).
- Check to make sure the package names are **edu.calpoly.android.lab4** (**src** folder) and **edu.calpoly.android.lab4.tests** (**test** folder).

Make sure that your project is targeting the latest version of Android but supporting API 10 and higher.

- In the Manifest file, find the **uses-sdk** XML component and change **android:targetSdkVersion** to the latest API version (17 at the time of writing this lab). If **android:minSdkVersion** is not set to 10, do so.

Change the author name in **strings.xml** to **<userid>**. Non-Cal Poly students can change this to an appropriate username handle of their choice.

If you haven't set up your project to use ActionBarSherlock (ABS), do so. If you already have the library project for ABS, you should be able to skip this step.

- Use the setup step from Lab 3 as a guide for downloading and creating the ABS library project and adding ABS as a library project to your imported stub project under **Properties** -> **Android** -> **Library**. You may need to remove the existing reference to ABS and replace it with a reference to the ABS project that you downloaded.

Run the following three Unit Test classes (located under the **test** folder) as Android JUnit Tests:

- **AdvancedJokeListTest2**
- **AdvancedJokeListAddFilterTest**
- **AdvancedJokeListAddFilterDeleteTest**

These Unit Tests are the same tests from the downloadable Lab 3 stub project. They should all pass. If not, double-check your code (if it's your old code from lab 3) and make sure it works as intended according to Lab 3.

1.2 Familiarize Yourself with the Source Code

The skeleton project that has been given to you for this lab contains a fully functional solution to Lab3. In particular, the AdvancedJokeList, JokeListAdapter, Joke, and JokeView classes are fully functioning classes from Lab 3. Additionally, many completed XML resource files (drawable, layout, menu) have been supplied as well. Most of the code in the new

It is a good idea, and good practice, to read through the source code. See how it compares to your implementation of Lab 3. It is especially important to do this if there were any parts of Lab 3 that you were not able to complete. The rest of the Lab will require you to update this source code to make use of Data Persistence so it is critical that you are familiar with it.

- Make sure the application runs and functions properly according to Lab 3. If you did not complete Lab 3, make sure you understand how the code base works, what State Lists are, how the Action Bar works, etc.

1.3 Fill in the Joke Class

You must fill in a few new areas of the Joke Class. Some new method stubs and a member variable have been added that you will have to fill in and make use of. Make sure not to delete or change these. In particular,
a member variable named **m_nID** has been added to the class which will contain the unique id assigned to the Joke from the Database:

- You must update the constructors to properly set **m_nID**:
  - For constructors that do not take in an ID parameter initialize m_nID to **0**.
- Comment out the current logic for the **equals(...)** method (the return statement).

- The **equals(...)** method now requires that only the ID values be equal for two Jokes to be equal.
- There is a getter and setter that needs to be filled in for **m_nID**.

Run the **JokeTest.java** Unit Tests to ensure that you have properly filled in this class.

Notice that if you try to run your application, it will not function properly. This is because the **equals(...)** method has been changed.

- Comment out the current logic for the **equals(...)** method, and uncomment the original **equals(...)** method logic. You will use your old **equals(...)** method logic until otherwise noted in the lab, just be warned that the **JokeTest** unit tests will fail unless you use the new logic.

Notice also that if you try to run the two **AdvancedJokeListAddFilter*Test.java** unit test files that passed previously, you will now get errors. This is because the **equals()** method has been changed. This is normal.

Note: *From here on out, **<u>there are no unit tests for this project</u>**. The overall functionality of the application is not changing at all, only the way information is saved and retrieved. However, you will be asked to demonstrate your application using a particular series of instructions that will test all attributes of the application for correctness. It will be obvious if the application is not working properly. But don't panic! This lab is complex, but a large portion of it will be studying and understanding code that will be provided to you in bits and pieces. Lab 4 is more about understanding components than coding them up.*

2. Maintaining Application State

You will now enable your application to maintain basic settings in the event that your Activity is destroyed and re-created, as well as maintain UI state across different runs of your application. In particular, the filter value, saved in **m_nFilter**, should be maintained if the Activity is destroyed and re-created for any reason. In this case the jokeList should be re-filtered to display only the jokes specified by **m_nFilter**. The filter value will be saved as Instance State. Additionally, the text in the m_vwJokeEditText will be saved and restored across separate runs of your programs via the Preferences mechanism. You can begin familiarizing yourself with the subject matter by reading the Android Documentation on Saving Persistent State in Activities.

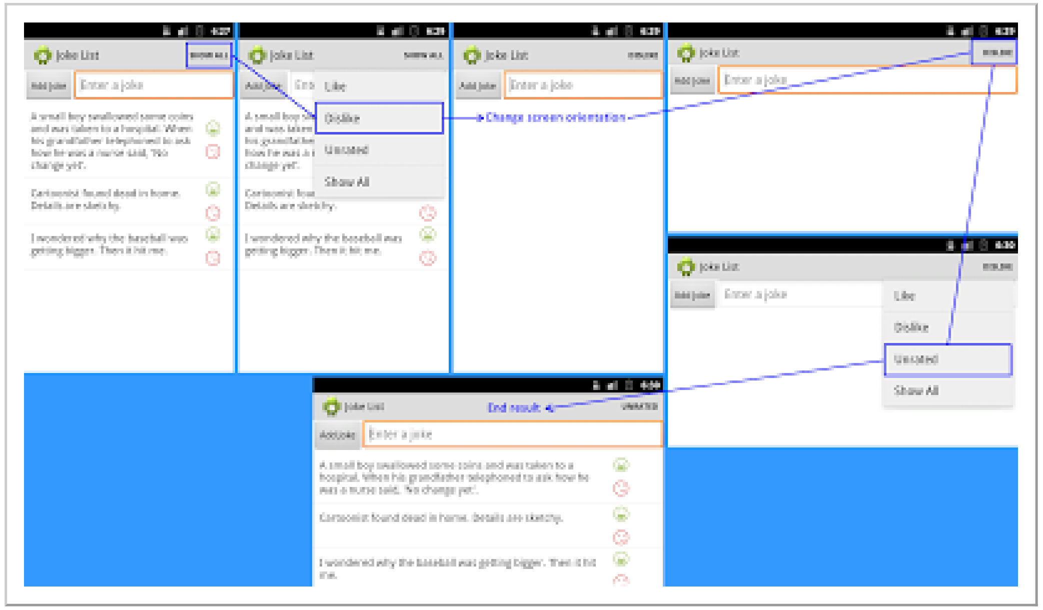
2.1 Instance State

Instance state is data that is private to a single instance of an Activity. This data is not shared across the application or made available to any other instances of the same Activity. Instance state can be stored in a Bundle object when the Activity becomes inactive, or destroyed and dumped from memory. The Bundle object is saved even though the Activity no longer resides in memory. When an Activity becomes inactive, its **onSaveInstanceState(...)** method is called. This is where saving of instance state happens. It is important to note that this only happens when the Activity is saved in the Application History stack, not when the Activity is closed or finalized.

The Bundle object is a map data-structure that uses key-value pairs to store data. The type of data that can be stored is limited to basic data types like primitives and strings. You can retrieve the data from the Bundle object when the Activity is re-created. The Bundle object containing the instance state can be accessed by overriding two separate methods. The first, which you are already familiar with, is the **onCreate(...)** method.   The second is the **onRestoreInstanceState(...)** method, which gets called immediately after the **onStart(...)** method. Both of these methods take a Bundle argument that contains the instance state if it was previously saved. For more information about the life cycle of an Activity, see the explanation and diagram here.

It is important to note that this Bundle object might not contain anything in the initial creation of the activity. The **onRestoreInstanceState(...)** method won't even get called on the initial creation of the Activity. Restoring instance state can be done in either method. However, using the **onRestoreInstanceState(...)** method instead of the **onCreate(...)** method has the benefit of logically separating initialization tasks from restoration tasks.

In the following two subsections you will be persisting the Filtering mechanism as instance state data. To start you will save the filter state of your AdvancedJokeList Activity in the **onSaveInstanceState(...)** method. You will then restore the state in the **onRestoreInstanceState(...)** methods. More background on this can be found in the Android Developer Guide on Application State and the Android Documentation on the Bundle Class. When finished, your application should function as depicted by the figure below (click to view full image):



Joke persistence is not yet implemented, this is expected.

2.1.1 Saving Instance Data

Begin by familiarizing yourself with the Android Documentation on the Activity.onSaveInstanceState(...) method. It is good to understand what the default implementation does before modifying it.

- Override the **Activity.onSaveInstanceState(Bundle outState)** method.
- Store the current value of **m_nFilter** in outState using the **SAVED_FILTER_VALUE** static constant string.
    - Use the appropriate **Bundle.put(...)** method.
- Call the super version of this method to ensure that other UI state is preserved as well.
    - *The default implementation of onSaveInstanceState(...) will save the state of any UI component in the Activity's content/layout hierarchy that has an id value. When you change the orientation of your device, this is why the state of basic UI components is remembered. If you fail to call the default super implementation, this will not occur.*

Of course, this means you have to change AdvancedJokeList to now properly modify **m_nFilter** during normal application usage in the code, including initially in **onCreate()**.

- You are left to complete this relatively trivial task on your own.

2.1.2 Restoring Instance Data

Begin by familiarizing yourself with the Android Documentation on the Activity.onRestoreInstanceState(...) method. It is good to understand what the default implementation does before modifying it.

- Override the **Activity.onRestoreInstanceState(Bundle savedInstanceState)** method.
- Call the super version of this method to ensure that other UI state is preserved as well.
    - *The default implementation of onRestoreInstanceState(...) will restore the state of any UI component in the Activity's content/layout hierarchy that has an id value. This is why when you change the orientation of your device the state of basic UI components is remembered. Things like the text in m_vwJokeEditText and which Joke is in expanded mode. If you fail to call the default super implementation, this will not occur.*
- Retrieve the value of **m_nFilter** from savedInstanceState using the **SAVED_FILTER_VALUE** key.
    - It is best to test the savedInstanceState parameter and the values you retrieve from it before using them. Check for null and use the Bundle.containsKey(String key) method.
    - Use the appropriate **Bundle.get(...)** method.
- Re-filter your joke list to ensure that the proper jokes are displayed. This filtering should happen the same way it does when the user selects a filter MenuItem from the Filter SubMenu.
    - In the skeleton project, the filtering is done with the **filterJokeList()** method.

2.1.3 Showing Persistent Filtering

The final task for this section is to make it clear to the app's user that the filter is being preserved. Currently there is no indication of filter preservation other than jokes appearing in and disappearing from the list when a filter is chosen. We will make it obvious when a filter is selected, as well as preserve that filter across destructive modifications such as orientation change, by changing the text on the Filter menu item itself whenever a filter is chosen. This shall be done via the **onPrepareOptionsMenu()** method.

- Create a new method of your name choice (such as **getMenuTitleChange()**) that returns the proper String depending on the current filter.
    - You can use **getResources()** to fetch the Strings for the menu item titles.

Familiarize yourself with the Android Documentation on the Activity.onPrepareOptionsMenu(...) method. It is good to understand what the default implementation does before modifying it.

- Override the **Activity.onPrepareOptionsMenu(Menu menu)** method.
- Change the title text of the Filter menu item, calling the method you just created above to get the proper text.
- Set the **m_vwMenu** variable appropriately after the title text gets changed.
- Call the **super** version of this method as a return statement to ensure that other menu state is preserved as well. Just like the previous overridden methods, you ***must*** call the super constructor.

Now the Filter menu item text will persist across destructive app modifications such as orientation change, but the title still needs to be updated when a new filter is chosen as well.

- Update **onOptionsItemSelected()** to also use the method you created above to change the menu's text depending on which filter gets chosen.

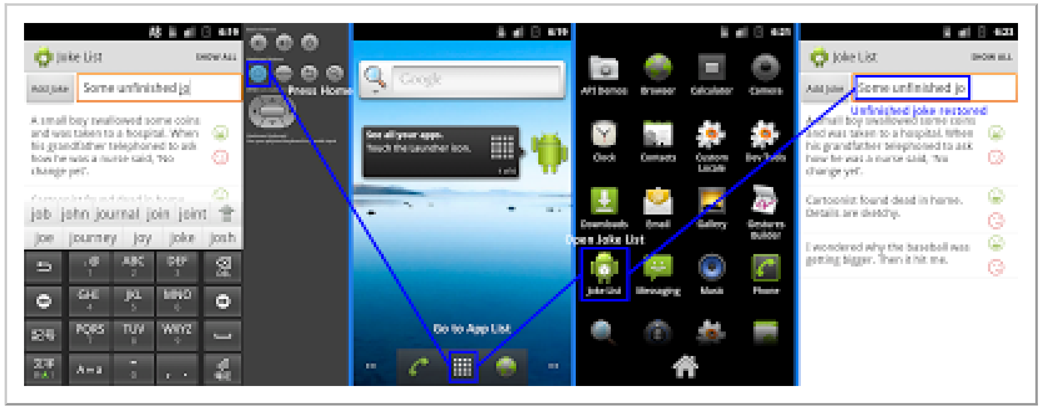
Try running your application. The default Filter should be **Show All** when the application starts up, as indicated by the Filter menu item's text.

- Select the **Filter** menu item (currently says **Show All**).
  - Select the **Dislike** MenuItem.
  - All the jokes should disappear since by default they are unrated at this point, and the menu text updated to say **Dislike**.
- Change the orientation of your device, which will force your Activity to be destroyed. In an AVD, use ***Shift + F11*** or ***Shift + F12*** (more AVD controls here).
  - No jokes should be displayed since the **Dislike** filter should have been saved, restored, and applied. The menu text should still say **Dislike**.
- *Note: It is expected at this time to have all jokes reset upon orientation change. We have not made the necessary changes to preserve the jokes yet.*

2.2 SharedPreferences

The SharedPreferences mechanism operates in a manner similar to saving Instance State in that primitive data is stored in a map of key/value pairs. The difference is that SharedPreference data can be shared across Application components running in the same Context and that the data persists across separate runs of the application. Alternatively, you can make the SharedPreferences data private to a single instance of an Activity.

In this next section you will persist the text in **m_vwJokeEditText** across multiple runs of an Application. This allows the user to work on a new joke across multiple sessions, and guarantees that if the process is killed for some reason, the user won't lose a joke they were working on. To start, you will override the default implementation of **onPause()** and save data in a private SharedPreferences object. You will then retrieve and restore the data in the **onCreate(...)** method. Saving your data in **onPause()** guarantees that if your process is killed, the data will still be available in a subsequent call to **onCreate(...)**. This is because **onPause()** is the earliest point at which an Activity can be killed by the system. You should be able to see this from the Activity Lifecycle Diagram. When finished, your application should function as depicted by the figure below (click image for full size):



2.2.1 Saving SharedPreference Data

Begin by reading the Android Developer guide on using SharedPreferences to save data, as well as the Android Documentation on the Activity.getPreferences(...) method and the SharedPreferences class. It is good to understand how these work before using them.

- Override the **Activity.onPause()** method and perform the following steps inside this method. Making sure to call the default implementation.
- Retrieve the private SharedPreferences belonging to this Activity by calling the **getPreferences(...)** method.
    - See the Documentation on Activity.getPreferences(...) for details on how to do this.
- Retrieve a SharedPreferences.Editor from the SharedPreference object.
    - See the Documentation on SharedPreferences.edit() for details on how to do this.
- Store the text in **m_vwJokeEditText** in the SharedPreferences by calling the appropriate **Editor.put...** method.
    - You should use **AdvancedJokeList.SAVED_EDIT_TEXT** as the key.

2.2.2 Restoring SharedPreference Data

The following should be done in the **onCreate()** method at the very bottom.

- Retrieve the private SharedPreferences belonging to this Activity.
- Retrieve the text that was saved in the onPause() method using the appropriate **SharedPreferences.get...** method.
    - The appropriate default value is an empty string **""**.
- Set the text in m_vwJokeEditText to the text you just retrieved from the SharedPreferences.

- Run your application to ensure that the text in m_vwJokeEditText is properly preserved across multiple runs of the application. See the figure in 2.2 above for details on how to test this.

Before you move on to joke persistence, you will change Joke.java to accommodate for this.

- Remove the current logic for **Joke.equals(...)** (the logic you used in Lab 3) and uncomment the new logic (ID comparison).

JokeTest should once again pass, but your application will not function properly. This is expected behavior as this issue will be resolved by the remaining sections.

3. Joke Persistence I: Database & Content Provider

Your ultimate goal at the end of this lab is to be able to persist Jokes in a SQLite Database. Due to Android's constant evolution this has become a difficult and widely undocumented subject at the time of modifying this lab, but the process will be broken down into small pieces with many code snippets involved to make it easier to digest.

We will work our way from creating the SQLite database inside of a content provider to creating the adapter that facilitates behavior between our content provider and our ListView, to reworking the ListView itself.

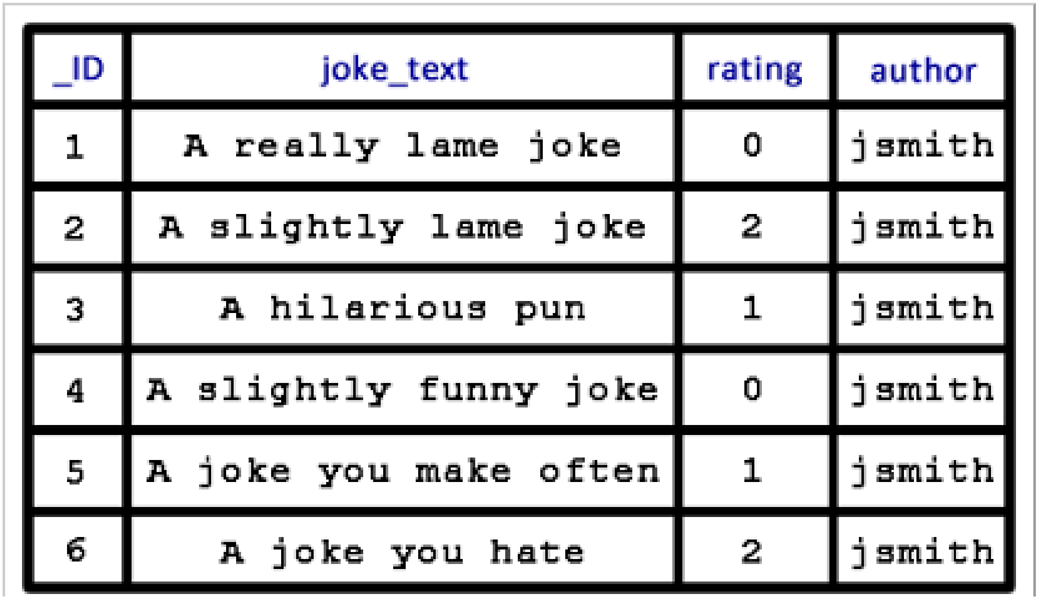
3.1 Content Providers

We'll begin with a discussion on Content Providers and how they relate to databases. The place to go for information on Android Content Providers is here. Use the Android page on Content Provider basics as a guide while reading this section.

What are Content Providers?

A content provider does exactly what it is named: It is a partner to a data source, such as a database of information, and provides content from that data source to various accessors. In the context of Android, this means that Android applications may access content providers for retrieval, modification or creation of data in the database. **A content provider is not a database**, but rather an organized representation of one**.** A database is an entity that applications want information out of, and the content provider acts as the abstract layer between the database and the application (similar to how an Adapter acts as the "glue" between a View and underlying data, like in Lab 3). We will be wrapping a database *inside of* a content provider interface so that we may achieve a single compound entity that both contains data and provides access to it. Our lab application is the accessor in this case.

Content providers expose the data from a data source using ***tables***. A table is a two-dimensional object that contains an organized representation of the data in a data source. The basics guide linked above contains an example table, but let's have another more relevant example (click image for full view):



_ID	joke_text	rating	author
1	A really lame joke	0	jsmith
2	A slightly lame joke	2	jsmith
3	A hilarious pun	1	jsmith
4	A slightly funny joke	0	jsmith
5	A joke you make often	1	jsmith
6	A joke you hate	2	jsmith

A table full of joke data contained in a content provider. Pretend this table is named "joke_table".

This is an example instance of the content provider table you will be creating. The resemblance to a list of jokes is striking: Each entry, or ***row***, in the content provider contains all Joke fields: an ID, the joke text, the joke rating and the joke author. In the above example, the rating is 0 for Unrated, 1 for Like and 2 for Dislike. This joke table serves as the content provider's organized source of data pulled from the database and is kept up to date through a set of operations that may be executed on the database. As you can guess, a table is an extremely effective way to get, modify and create data in a database: Why would we want to deal with an unorganized database, much less force it to organize itself? Moreover, a table is clean and easy for both humans and machines to read. Keep in mind that **a table is not a content provider** but rather a part of one, and **a table is not a database** but rather an organized representation of part of one.

To put it simply: We will store our joke data in a database, and that database will be inside of a content provider. That content provider will expose the database's data in an organized table to our application. Finally, our application will have access to create, modify and obtain the data exposed by the content provider, and the content provider will handle any modifications requested by our application and update the database accordingly.

3.2 The SQLite Database

With a brief background on content providers and tables, we will now focus on the aspects of the actual database. First, you guessed it...more information!

What are Content URIs?

A ***Content URI***, referred to as just **URI** throughout the remainder of this lab, is essentially a special code that is required when trying to perform an operation through a content provider. First, let's look at examples of the two Content URI formats we will use in this lab:

```
content://edu.calpoly.android.lab4.contentprovider/joke_table/filters/1

content://edu.calpoly.android.lab4.contentprovider/joke_table/jokes/5
```

While similar in appearance to a URL, a URI contains two important pieces: an address to the content provider, and a statement with a specific intent that is parsed by the content provider using a ***URIMatcher*** (more on this later). All Content URIs in Android are formatted using the "content" scheme, which is a fancy term that means "prefix the URI with the text `content://`".

In both URIs displayed above, edu.calpoly.android.lab4.contentprovider is the ***authority***. This is a fancy term for "match me with the content provider at this location". Not to anyone's surprise, we will implement our own content provider in this lab so it makes sense to point to it when trying to invoke operations on it. Rocket science, right? Everything else that follows the authority is referred to as the ***path***.

In both URIs displayed above, the next part of the path is joke_table. This part means "this is the table I'm trying to access". Since our content provider will have only one table, it makes sense to point to it when trying to do anything with it.

The remainder of the paths of the above URIs, filters/1 and jokes/5, are basically keys for the operations we wish to perform. These can be whatever we want them to be, but when they are passed to the operations they must be expected, as you'll see soon. The first one indicates that we want to include the Like filter on our operation, since Joke.LIKE has a value of 1. You'll notice in the joke table image in 3.1 that the rating column uses 1 for Like. The second one indicates that we want to include the 5th joke in the table on our operation. Using URIs, we can specify different behavior within the same operation. **Note:** The reason why we need to have two separate paths for filters and jokes is because both of them refer to numbers. If they were left as just numbers (1 and 5) then there would be no way to differentiate filters from joke IDs.

See more detailed information about Content URIs here.

What Operations Can be Performed Through the Content Provider?

The content provider will provide four operations that may be performed on data from the database: **query**, **insert**, **delete** and **update**. As you can probably guess, these are operations that we will implement in our JokeContentProvider class as methods.

The operations are briefly described below at a high level:

Query - Given a URI containing a numerical Joke filter and conditions for which rows to fetch, the content provider returns a ***cursor*** (more on this later) that contains a list of rows that match the conditions specified. For example, given the table above, a query made for jokes that have a rating of 0 would return a cursor object that contains two rows: the row with ID 1 and the row with ID 4.

Insert - Given a URI containing a Joke ID and list of values, the content provider inserts a new object into the database and updates the table with the new information. The inserted object appears as a new row at the bottom of the table, populating the row's columns with the list of values passed in. Then the content provider returns the ID of the newly inserted row. We do not need to worry about passing in a proper ID for insertion because we will create the database with a statement that will automate the ID assignment system for us. Convenient!

Delete - Given a URI that contains a Joke ID, the content provider locates a row in the table with a matching ID and purges it from both the table and the database. The content provider then returns the number of rows that were deleted (in our case, only one row will be deleted per operation).

Update - Given a URI that contains a Joke ID and a list of values, the content provider searches for a row with the matching ID and updates that row's columns with the passed-in values. Then the content provider returns the number of rows that were updated (in our case, only one row will be updated per operation).

3.2.1 JokeTable.java

All right, it's finally time to get our hands dirty! If the past few steps were an information overload, don't worry: you can refer back to them if you need to check terminology or get your bearings straight.

We need to create two special helper classes that will aid our Content Provider class first. The first is JokeTable.

Open up **JokeTable.java**. Spend some time looking at the code, particularly the class variables. You will see some familiar terms such as **joke_table**, which is the name of the table in which we will operate on all of our jokes. You will also see strings that represent the columns for each row in the joke table that match with member variables in the Joke class. There is an **_id** field, but as explained above we will automate row ID creation in our database. In fact, look at the **DATABASE_CREATE** string: Even if you don't know any SQLite syntax, you can probably understand what this string is doing and what it's meant for.

- Fill in the **onCreate(...)** method. This method will help open the database and create the table **joke_table**.
    - Make a call to the database's **execSQL(...)** method, passing in the database creation statement.
- Fill in the **onUpgrade(...)** method. This method will help upgrade the database, but we are not overly concerned about this operation.
    - Just to send a warning that the database is being upgraded, send a Warning Log message to Logcat. Pass in `JokeTable.class.getName()` as the tag, and then create a message that explains that the database is being upgraded from the old version to the new version for the second **Log.w(...)** argument.
    - Make a call to the database's **execSQL(...)** method, passing in the

        appropriate String. We want to get rid of the table since the database is being upgraded.

    - Finally, call this class' own **onCreate(...)** method.

3.2.2 JokeDatabaseHelper.java

The second helper class we will need is **JokeDatabaseHelper.java**. This class will have some assistance from JokeTable to actually open and minimally manage the database for our content provider. The content provider will naturally contain a copy of the database. JokeDatabaseHelper contains a reference to the database as well as the starting database version.

- Make JokeDatabaseHelper extend **android.database.sqlite.SQLiteOpenHelper**. This class is essential for managing our SQLite database from within the content provider.
    - In the constructor, make a super call, passing in the appropriate parameters.
        - **null** is an acceptable value for the CursorFactory parameter, signalling default behavior.
- For the two required overridden methods **onCreate(...)** and **onUpgrade(...)**, simply make calls to the respective static methods in JokeTable.

And...that's it. The SQLite database creation and management is prepared for the content provider.

3.3 The Content Provider

We will now implement the content provider. As a reminder, this bad boy will wrap around our SQLite database and give other classes an easy way to interact with it. All coding in this section will take place inside of **JokeContentProvider.java**.

Open up the class and take a look at its class variables and methods. You will see the four database operations mentioned above. Implementing those methods will be the bulk of this step.

- Observe the class member variables. Guess what most of them deal with? If you guessed URIs, then you are correct! There is one variable that is different (a reference to the database) but otherwise much of this class deals with handling URIs inside of the four database operation methods.

  - Make sure you understand what these URI-related variables mean on a general scale. If you need a refresher on URIs, scroll up to a previous section in this lab that describes URIs.

- One of the variables is a ***URIMatcher***. A URIMatcher is a class that matches URIs passed in to database operation methods and allows for different variations of the same operation to be made at any given time. In Object-Oriented Programming, one of the design patterns is the Strategy pattern. The URIMatcher follows the Strategy pattern very closely: it allows for the same operation to be performed with different algorithms.

  For example, we can request a **query** operation on the database by calling **query(...)** from our application and, using a URIMatcher, analyze the URI being passed in. The URIMatcher checks for proper, expected URIs. What is the URI's authority? What is the URI's path? These are questions that the URIMatcher answers by checking against a list of URIs it holds. In our example, we could perform a query on our joke table in two different ways using two different URI forms:

```
content://edu.calpoly.android.lab4.contentprovider/joke_table/filters/2

content://edu.calpoly.android.lab4.contentprovider/joke_table/jokes/3
```

Assuming our URIMatcher recognizes both of these as properly formed URIs (by matching them to URI formats it is given upon initialization), we could theoretically use a `switch` statement inside of our **query(...)** method in the content provider to catch both possibilities. If it matches the first URI format, we can modify our query to fetch all table rows that have the rating "Dislike" (since the Dislike filter is represented by the integer value '2'). If it matches the second URI format, we can modify our query in a separate switch case to instead fetch all table rows that have the ID 3 (which will only return 1 row). This means that we have absolute flexibility when it comes to querying the database (if this is confusing to you, it will become clear once you finish filling in the **insert(...)** method and implement the **query(...)** method).

Observe the **sURIMatcher** variable's initialization. See how it adds a single URI format to its list of acceptable URI formats:

```
sURIMatcher.addURI(AUTHORITY, BASE_PATH + "/jokes/#", JOKE_ID);
```

This URI format looks for URIs formatted as `content://<authority>/<basepath><path>`, where:

- `<authority>` is `edu.calpoly.android.lab4.contentprovider`
- `<basepath>` is `joke_table`
- <path> is `jokes/#`

When all of the parts are put together, that URI format looks like this: `content://edu.calpoly.android.lab4.contentprovider/joke_table/jokes/#`
In other words, the URIMatcher is ready to accept URIs that deal with jokes and provide a joke ID. It matches that URI format to the value **JOKE_ID**, which is the value that the URIMatcher will return after finding a match. This allows us to provide special logic inside of the operation methods to alter the actual database operation calls before we make them (if this is confusing to you, look at the **insert(...)** method and observe the case inside the switch statement).

  - Add a second URI format to the URIMatcher, this time for URIs that deal with filters and provide a filter ID. Make the URIMatcher correspond that URI format with the **JOKE_FILTER** value.

  - *For more information on URIMatcher, read up on it here.*

- Fill out the **onCreate()** method.

  - Initialize the database class variable as a new JokeDatabaseHelper, passing in the appropriate parameters.

    - You will need to pass a couple of constants from JokeDatabaseHelper into the constructor.
    - You can use getContext() to retrieve the context for the context parameter.

Boom, your SQLite database is now open and ready for business. Such an easy way to manage a SQLite database, isn't it? That's the power of a SQLiteOpenHelper.

3.3.1 Query() and Cursors

The first of the content provider operations we will implement is **query(...)**. What this method will do is format a database query with special arguments to indicate which rows we want to get out of our joke table, and then perform the actual query call on the database itself. You will be walked entirely through this method as it is rather complex and introduces a new data type: ***Cursors***.

Below is the entire solution to the **query(...)** method. Copy and paste it over the **query(...)** method in JokeContentProvider.java (just under the method's comment block):

```
     @Override
 public Cursor query(Uri uri, String[] projection, String selection,
 String[] selectionArgs, String sortOrder) {

 /** Use a helper class to perform a query for us. */
 SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

 /** Make sure the projection is proper before querying. */
 checkColumns(projection);

 /** Set up helper to query our jokes table. */
 queryBuilder.setTables(JokeTable.DATABASE_TABLE_JOKE);

 /** Match the passed-in URI to an expected URI format. */
 int uriType = sURIMatcher.match(uri);

 switch(uriType) {
 case JOKE_FILTER:

 /** Fetch the last segment of the URI, which should be a filter number. */
 String filter = uri.getLastPathSegment();

 /** Leave selection as null to fetch all rows if filter is Show All.
Otherwise,
  * fetch rows with a specific rating according to the parsed filter. */
 if(!filter.equals(AdvancedJokeList.SHOW_ALL_FILTER_STRING)) {
 queryBuilder.appendWhere(JokeTable.JOKE_KEY_RATING + "=" + filter);
 } else {
 selection = null;
 }
 break;

 default:
 throw new IllegalArgumentException("Unknown URI: " + uri);
 }

 /** Perform the database query. */
 SQLiteDatabase db = this.database.getWritableDatabase();
 Cursor cursor = queryBuilder.query(db, projection, selection, null, null,
null, null);

 /** Set the cursor to automatically alert listeners for content/view
refreshing. */
 cursor.setNotificationUri(getContext().getContentResolver(), uri);

 return cursor;
 }
```

- Study this code. We will now cover it in small pieces:

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder)
```

Above is the method signature. The comment block already provided for you explains the method and how to get more information about the **query(...)** method JokeContentProvider is overriding.

The parameters:

uri - As discussed previously in the section on URIs, this contains information about how to tweak the operation (a database query) before actually making it.

projection - This is the set of column names (think back to **JokeTable.java**) that we are going to place into the Cursor we will return. Just assume we will always place all of the columns (_id, joke_text, rating, author) into the Cursor for simplicity's sake. Cursors are essentially lists that contain any table rows returned after making a database query.

selection - This is the most important part of each operation in our content provider, as it defines how to format each database operation. It is also referred to as the **WHERE clause**. For example, in the above code selection is being altered to have the statement `JokeTable.JOKE_KEY_RATING + "=" + filter`. Assuming filter is Like, for example, this translates into the raw String `rating=1`. This means that when the actual call to query is made near the bottom of the **query(...)** method, it will probe the joke table for all rows that contain the rating 1 and return them inside of a cursor. Referring back to the joke table image in section 3.1, this means that the query will return a Cursor with row 3 and row 5, since those contain a rating of 1. ***Important note:*** *if selection is set to null, then all rows will be returned.*

We aren't concerned with the rest of the parameters.

```
    /** Use a helper class to perform a query for us. */
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
```

We are using a SQLiteQueryBuilder to construct our query for us. You'll notice that there's a call made to **appendWhere(...)** later on in the method. This will automatically add whatever is passed into **appendWhere(...)** onto the end of our **selection** variable when the query is finally being made. We don't even have to make any changes to **selection**, the SQLiteQueryBuilder does it for us!

```
    /** Make sure the projection is proper before querying. */
checkColumns(projection);
```

Speaking of the **checkColumns(...)** method, here's the whole thing provided to you:

```
        private void checkColumns(String[] projection) {
    String[] available = { JokeTable.JOKE_KEY_ID, JokeTable.JOKE_KEY_TEXT,
   JokeTable.JOKE_KEY_RATING,
    JokeTable.JOKE_KEY_AUTHOR };


    if(projection != null) {
    HashSet<String> requestedColumns = new
   HashSet<String>(Arrays.asList(projection));
    HashSet<String> availableColumns = new
   HashSet<String>(Arrays.asList(available));


    if(!availableColumns.containsAll(requestedColumns)) {
    throw new IllegalArgumentException("Unknown columns in projection");
    }
    }
    }
```

What this does is perform a sanity check to make sure that the projection values match the expected values. Since we're always going to pass in all columns into our projection, and this method checks all columns, we aren't worried about anything. Don't concern yourself too much with this method.

```
/** Set up helper to query our jokes table. */
queryBuilder.setTables(JokeTable.DATABASE_TABLE_JOKE);
```

This line makes sure that we are going to query the correct table. In this case, that would be `joke_table`.

```
      /** Match the passed-in URI to an expected URI format. */
  int uriType = sURIMatcher.match(uri);
```

```
switch(uriType) {
case JOKE_FILTER:

/** Fetch the last segment of the URI, which should be a filter number. */
String filter = uri.getLastPathSegment();

/** Leave selection as null to fetch all rows if filter is Show All. Otherwise,
 * fetch rows with a specific rating according to the parsed filter. */
if(!filter.equals(AdvancedJokeList.SHOW_ALL_FILTER_STRING)) {
queryBuilder.appendWhere(JokeTable.JOKE_KEY_RATING + "=" + filter);
} else {
selection = null;
}
break;

default:
throw new IllegalArgumentException("Unknown URI: " + uri);
}
```

Here is where our URIMatcher goes to work for us. Our content provider's **query(...)** method is only checking for the URI format that matches JOKE_FILTER (which looks for URIs formatted like this: `content://edu.calpoly.android.lab4.contentprovider/joke_table/filters/#`). This means that we are only querying for rows that contain a certain joke rating. This reflects what our original application does via its filtering system.

Note that **getLastPathSegment()** fetches the last part of the path in the passed-in URI variable, which in this case is the number at the very end of our URI. This corresponds to a joke rating, and we are ordering the query builder to set the selection to this joke rating.

```
/** Perform the database query. */
SQLiteDatabase db = this.database.getWritableDatabase();
```

```
    Cursor cursor = queryBuilder.query(db, projection, selection, null, null,
    null, null);
```

Finally, the above code shows the actual query operation is carried out on the database! This is where we obtain our Cursor, which will contain rows in the joke table whose *rating* value matches whatever filter value we obtained from the URI and placed into our selection statement. This is the "wrapped" operation call that we've been talking about, the call made straight to the database. We are inside the content provider's **query(...)** method, which is the wrapper for the actual database operation call. Hopefully, the big picture is more clear at this point.

```
        /** Set the cursor to automatically alert listeners for content/view
        refreshing. */
    cursor.setNotificationUri(getContext().getContentResolver(), uri);


    return cursor;
```

Content Resolvers act as event resolvers for any accessors that are working with content providers, such as our AdvancedJokeList application. They notify observers of underlying data changes which triggers content refreshing. Our ListView will become an observer because it will eventually use a JokeCursorAdapter, which will bind the ListView to a Cursor. And since the cursor is attached to the joke table and we are registering our application's content resolver to changes made with the given uri (meaning any change made by the query will set the content resolver off), any changes that are made to the joke table are automatically resolved in our ListView. This is extremely efficient, since this means less updating work on our end!

Finally, we return our cursor.

Phew, that was a lot of information to digest! Now let's do some more coding.

3.3.2 Insert()

The content provider's **insert(...)** method is a wrapper for an actual database **insert** operation call, just like how **query(...)** is a wrapper for the **query** database operation call. It also looks very similar to **query(...)** implementation-wise. This method is almost entirely implemented for you.

- Replace the null assignment to **sqlDB** with a proper statement that opens the database as writable.
    - Hint: there's a line just like it in **query(...)**.

To explain this method in words:

- The database gets opened for writing.
- The URIMatcher checks the passed-in URI for a URI format that implies a joke and contains the joke's ID. We don't want to insert anything if the URI implies filtering and contains a filter's value, hence why `case JOKE_FILTER` is left out.
- Insert the joke into the database.
- Set the ContentResolver to notify components attached to the joke table (refreshing the ListView once we give it a CursorAdapter and bind its JokeViews to a Cursor). This will automatically go off because we just made a change to the joke table--we inserted a new joke.
- Return a URI that contains the ID of the joke we just inserted into the joke table. Our database was created with a statement that automatically increments and assigns IDs to rows, so the ID being returned in the URI will be utilized by AdvancedJokeList later to set each Joke's ID variable properly (remember at the very beginning of the lab in Joke.java, we set the ID of each Joke to 0 when it wasn't initialized and we didn't care about it? This is why).

3.3.3 Delete()

The content provider's **delete(...)** method is a wrapper for an actual database **delete** operation call, just like how **insert(...)** is a wrapper for the **insert** database operation call. It will look similar to the two wrapper methods you've already completed so far.

Fill out this method as follows:

- Open the database for writing, just like in the other wrapper methods.
- Declare an integer variable that will keep track of the number of rows that get deleted when the database **delete** operation is made, initializing it to 0.
- Using the URIMatcher, match the URI format denoted by JOKE_ID.
    - If the URI format matches JOKE_ID, obtain the last path segment in the URI and store it in a String. You've done this in the **query(...)** method.
    - Perform the **sqlDB.delete(...)** call, setting the return value to the appropriate variable.
        - You created a WHERE clause in **query(...)** before, but that one was created for the filter value and you passed it to the query builder. This time, create one for the joke ID and pass it in as the second parameter to **sqlDB.delete(...)**.
        - Pass in **null** for **whereArgs**.
- Set the ContentResolver to notify components attached to the joke table using **notifyChange(...)**.
    - This should be done the same way as it is done in **insert(...)**.
    - This should only be done if the number of rows deleted is greater than 0.
- Return the number of rows deleted.

3.3.4 Update()

The content provider's **update(...)** method is a wrapper for an actual database **update** operation call, just like how **delete(...)** is a wrapper for the **delete** database operation call.

You are tasked with implementing this method yourself, but it is pretty much identical to the implementation for **delete(...)** only with a different database call.

Congratulations, you have successfully wrapped a SQLite database inside a Content Provider! Now we have to make the provider available to our application:

- In the Android manifest file, add the following inside the <application></application> tags but after the <activity></activity> tags:

```
<provider android:name="edu.calpoly.android.lab4.JokeContentProvider"
          android:authorities="edu.calpoly.android.lab4.contentprovider">
</provider>
```

Note: Because we are only keeping this content provider local to our application, there are few permission settings to worry about. However, when making a content provider open to other applications besides the one the provider belongs to, there is a need to worry about permissions. Read more about it here.

Now we will be moving on to the next step, which is implementing the CursorAdapter and OnJokeChangeListener.

4. Joke Persistence II: OnJokeChangeListener & CursorAdapter

As a summary, let's go over the path of data flow for our application so far:

1. Data is modified, created and retrieved from the database through the JokeContentProvider.
2. The JokeContentProvider exposes the database's data in a table.
3. A Cursor may be retrieved from the table via a **query(...)** call in JokeContentProvider.
4. The retrieved Cursor contains rows from the table that have a rating that matches the filter ID passed in to **query(...)** via the URI parameter.

Now we will add another step: the Cursor will be bound to a ListView through the JokeCursorAdapter.

With the database and content provider all set up, we will now implement the CursorAdapter and complete the bridge between database and ViewGroup. The JokeCursorAdapter will function similarly to the previously used JokeListAdapter in Lab 3, but this adapter is binding a ListView to a Cursor that is tied to the joke table of the content provider we just implemented.

But before we handle that, we will implement functionality for listening to JokeView changes. The JokeCursorAdapter also contains an **OnJokeChangeListener** which will complete the cycle of database-to-ListView-to-database updating (thanks also in part to the ContentResolver set in our content provider, as you will find out in the next section).

What's this OnJokeChangeListener for, exactly?
In short, it has to do with Joke change preservation going back to the database. When the list of Jokes is displayed on-screen, Jokes are retrieved from a cursor and then wrapped into JokeViews. The JokeViews let the user modify the internal state of the Joke, in particular the ratings for a joke. Once the user has modified the rating for a Joke, the Joke object stops being an exact copy of the data in the database. If we want to make sure that the new rating gets preserved we must be able to write that rating change back to the database. You implemented this functionality in the previous lab, but not as a database.

The question is then whose job is it to write that change back to the database? Is it the job of the Joke object, the JokeView object, the Adapter, or the Activity itself? The JokeView and the Adapter are two very specialized classes, charged with specific tasks that don't really care about the persistent state of a Joke. The Adapter's job is to provide View objects for Jokes, and the JokeView's job is to show what a Joke looks like and provide controls for manipulating the internal state of the Joke.

That leaves two possibilities, the Joke and the Activity. While it is entirely acceptable to put the responsibility of persisting state on the Joke object, in this case we are going to follow the Model-View-Controller pattern and place that responsibility in the hands of the Activity. By doing this, we decouple the Joke class from the idea of a Database entirely. This allows the same Joke class to be used with or without a database (evidence of this is Lab 3 and this lab). But now we need some way for the

JokeView to signal to whomever is responsible, that its Joke object has changed and needs to be written back to the database.

The JokeView is just going to put its foot down and say: "Whoever you are who is responsible: if you want to know when the Joke changes, then you've got to tell me. I'll let you know which method I'm going to call to notify you."

Thus, a static interface named **OnJokeChangeListener** has been added to the JokeView class. This interface specifies a callback method that gets called when a Joke's internal state changes. Additionally, each JokeView class now holds a single member variable reference to an OnJokeChangeListener. Any class that is interested in receiving notifications that a Joke has changed should implement the OnJokeChangeListener and register itself with the corresponding JokeView.

Open up **JokeView.java:**

- In the constructor, set the **m_onJokeChangeListener** variable to **null**.
    - *We only want to listen for joke changes when the actual joke values change (such as the Joke rating), not when a new Joke object is being assigned to the JokeView. This is why we aren't listening for changes yet.*
- Fill in the two new methods for setting and getting the OnJokeChangeListener.
    - Fill in **setOnJokeChangeListener(...)**.
    - Fill in **notifyOnJokeChangeListener()**:
        - Make sure to only call **onJokeChanged(...)** when the listener isn't null, otherwise there will be an exception thrown.

Now it's the CursorAdapter's turn.

Open up **JokeCursorAdapter.java** and examine the code.

- Fill out the **setOnJokeChangeListener(...)** method.

- Make JokeCursorAdapter extend **android.support.v4.widget.CursorAdapter**.
    - In the constructor, uncomment the super call.
    - Fill in the **bindView(...)** method.
        - *This method takes an existing view and populates it with underlying data from a cursor.*
        - You will need to extract a Joke out from the cursor. You can easily do this using Cursor's **getX(...)** methods to construct a new Joke object.
            - Remember those column IDs in JokeTable, the ones that correspond to the locations of each column of a row in the joke table? They also function as indices into each column of a row in a Cursor, since both the Cursor and the joke table have the same row layout.
        - Set the view's OnJokeChangeListener to **null** to avoid refreshing the data when the joke data is not actually refreshing.
        - Set the view's Joke reference properly.
        - Set the view's OnJokeChangeListener to **m_listener**.
            - *Attaching this listener to each JokeView will set us up for easy refreshing in the next section.*
    - Fill in the **newView(...)** method.
        - *This method creates a new view and populates it with underlying data from a cursor.*
        - This is implemented similarly to **bindView(...)**, but requires that you create, initialize and return a new JokeView instead of an existing one.

- *Notice any similarities between the functionality of JokeCursorAdapter compared to JokeListAdapter?*

Great, now we're almost done. We just need to set up AdvancedJokeList to use our database and implement the OnJokeChangeListener.

5. Joke Persistence III: AdvancedJokeList & Loaders

The final step in this lab is to update AdvancedJokeList so that it completes the cycle of data transfer to and from the database and ListView children. Now we just have to integrate cursors and the application should work completely.

- Visit the CursorAdapter documentation page and read the documentation for the first constructor. Uh oh! It's deprecated, meaning it's not a good idea to use it. Fortunately, we avoided this in our implementation, but why was it a bad thing to begin with?
    - The former way of managing cursors was to use **requery()**, **startManagingCursor()** and **stopManagingCursor()**. The problem with this is that these methods are called on the same thread that the rest of the application's user interface sits on, and this could cause slow application responses and sometimes even dead hangs in execution!

We initialized our CursorAdapter appropriately, but we need to integrate cursors into AdvancedJokeList. We can't use the above methods to manage it. Fortunately, there's good news in the form of a newer class: **Loaders** gets around the previous method's issues by using asynchronous handling, and as a bonus they manage their loaded data automatically for us. Conveniently, there is a CursorLoader class.

We are going to make AdvancedJokeList use the LoaderManager class to load and manage our CursorLoader asynchronously, and the CursorLoader class to load and handle cursors asynchronously.

- *Just what is a Loader? Find this out and more here.*

5.1 The Loaders

Here we go! First up is the LoaderManager.

In **AdvancedJokeList**:

- Make the class extend **SherlockFragmentActivity** instead of SherlockActivity, otherwise we can't use a LoaderManager.
- Make the class implement **android.support.v4.app.LoaderManager.LoaderCallbacks<Cursor>**.
    - Add the unimplemented methods, but refrain from filling them out for now.
- In **onCreate()** after your adapter is being initialized, make a call to **getSupportLoaderManager()** and call its **initLoader(...)** method.
    - There is conveniently a **LOADER_ID** class variable meant to be used for initializing the CursorLoader through the LoaderManager.
    - Pass in **null** for the Bundle.
    - Pass in **this** for the LoaderCallbacks<Cursor>, since you made AdvancedJokeList implement that exact interface.
    - *This method call will initialize whatever loader we want in the **onCreateLoader(...)** method, but we now need to specify that it is indeed a CursorLoader being loaded.*

More information on the LoaderManager and how to implement the callback methods can be found here.

Next is the CursorLoader (that's the main documentation page, but make sure to import the support version in AdvancedJokeList).

in **AdvancedJokeList**:

- Fill in **onCreateLoader(...)**:
    - You thought you were done with SQLite and databases? Think again! **onCreateLoader(...)** will use the LoaderManager's magical powers to initialize a Loader for us, but the CursorLoader requires a Content Provider. This means we will do something similar to what we did earlier.
        - Create an array of Strings to serve as a projection for the CursorLoader.
            - Once again, JokeTable's column names will come in handy here. Place them into the projection in the order they appear in JokeTable.
        - Create a URI and initialize it using **URI.parse(...)**.
            - You want to create a URI formatted for filters similar to how you did when doing so for the content provider, but you'll also need to get the rating that corresponds to the currently selected filter and append it at the end of the URI String. Obtain this rating any way you want.
                - It is recommended that you create a helper method that, depending on the currently selected filter (**m_nFilter**), returns the corresponding rating as a String (e.g., if the currently selected filter is **FILTER_UNRATED**, you should return "" + **Joke.UNRATED**). Remember that Like = 1, Dislike = 2 and Unrated = 0.
                    - If the current filter is Show All, this method should return something that is not equal to the numerical rating values of Like, Dislike or Unrated (you can use **SHOW_ALL_FILTER_STRING**, which is set to the String "4" by default). You may also change the way **JokeContentProvider.query(...)** compares filter

values so that it doesn't use SHOW_ALL_FILTER_STRING, but you need to keep the selection null to retrieve all rows if Show All is the current filter.

        - Create a new CursorLoader and initialize it with the uri and projection you just created.

            - *The CursorLoader will load the Cursor after making a hidden automated query call. This is difficult to notice, but if you think about it, the Cursor needs to come from somewhere...*

- Change the type of **m_jokeAdapter** from **JokeListAdapter** to **JokeCursorAdapter**.

    - Also change how it is constructed in **onCreate()**.

        - Since there is no cursor to give it yet (LoaderManager and CursorLoader will do this later), it is perfectly acceptable to pass null as the Cursor here.

        - Pass 0 as the flags parameter. For more details on why we are doing this, read the documentation for CursorAdapter.

- Fill in **onLoadFinished(...)**:

    - Call your adapter's **swapCursor(...)** method, passing in the now fully-loaded Cursor.

        - ***swapCursor(...)*** *is new to API 11 (but accessible in older versions via the support library). There is a similar method,* ***closeCursor(...)****, but since we are using LoaderManager and CursorLoader all the closing and opening is handled for us!*

    - *This method is guaranteed to be called after* ***onCreateLoader(...)*** *is* *called and the Loader finishes loading whatever it is tasked to load.*

- Fill in **onLoaderReset(...)**:
    - Call your adapter's **swapCursor(...)** method, but pass in **null** since the loaded object is about to be reloaded and therefore the data behind it needs to be invalidated.

5.2 Cursor & OnJokeChangeListener Integration

Time to take out the trash!

- Remove the **m_arrFilteredJokeList** and **m_arrJokeList** variables entirely and delete all code relevant to them.
- Remove all code from inside of **addJoke(...)**, **filterJokeList(...)** and **removeJoke(...)**, but do not delete those methods entirely.
- Remove the initialization and addition of the preloaded jokes in **onCreate()**.

Out with the old, now in with the new:

- Create a void method called **fillData()** that takes no parameters:
    - This method should restart the CursorLoader and reset **m_vwJokeLayout**'s adapter, effectively refreshing the JokeViews and their corresponding values in the database.
        - Make a call to **getSupportLoaderManager()** and invoke its **restartLoader(...)** method. This will refresh the cursor in your JokeCursorAdapter automatically. You don't have to even lift a finger!

            - now that your adapter has a new Cursor, set your ListView's adapter to **m_jokeAdapter** again. This combined with the next method will complete the data refresh cycle.
            - *You want to call this method whenever there are changes made to any of the jokes in the list, such as a rating being changed or a new joke being added.*

Now you will finally invoke a majority of the content provider database operation wrapper methods whenever a joke is updated, added, removed or a new filter is chosen.

- Have AdvancedJokeList implement **OnJokeChangeListener**.
    - Fill in the **onJokeChanged(...)** method:
        - Create a new URI variable that will tell the content provider's URIMatcher in **update(...)** that the row with **joke.getID()** needs to be updated.
        - Create a new ContentValues variable and use **ContentValues.put(...)** to place the joke text, rating and author inside of it.
            - Remember that these will be the new values to be added to the database for the designated joke. See **JokeContentProvider.update(...)** for more details.
        - Call **getContentResolver().update(...)**, passing in the URI and

ContentValues variables where appropriate and **null** for all other parameters.

        - Before updating the list of JokeViews, you must set the OnJokeChangeListener for **m_jokeAdapter** to **null** otherwise the application will loop infinitely due to the OnJokeChangeListener both receiving news of a change and sending one itself.

            - Now you must also reset **m_jokeAdapter**'s OnJokeChangeListener in **onLoadFinished(...)** back to **this**.

        - Now make a call to **fillData()** to complete the refreshing cycle. Now whenever a value changes in a JokeView the change will immediately sync back to the database.

    - In the **onCreate()** method, set the OnJokeChangeListener of your adapter to **this**.

    - In **JokeView.java**, you must now call **notifyOnJokeChangeListener()** whenever you change any of the underlying joke data on-screen.

        - The only time this happens is when the joke rating changes in **onCheckedChanged(...)**.

- Update **removeJoke(...)**:

    - Create a new URI variable that will tell the content provider's URIMatcher in **delete(...)** that the row with joke.getID() needs to be deleted.

- Call **getContentResolver().delete(...)**, passing in the URI where appropriate and **null** for all other parameters.
- Make a call to **fillData()** to complete the refreshing cycle.
- Unfortunately, because your ListView is now bound to a completely different object you can no longer use the old method of retrieving a Joke to remove or else you will get a ClassCastException. This means that we will need to expose a bit of JokeView's data to the application.
    - Update **JokeView.java**:
        - Create a public getter method called **getJoke()** that returns the JokeView's underlying Joke object.
    - Update **AdvancedJokeList.java.onActionItemClicked(...)**:
        - Change the call to **removeJoke(...)** so that it retrieves the Joke from the currently selected JokeView.
            - Do this using **getJoke()** and **m_vwJokeLayout.getChildAt(...)** passing in the currently selected position.

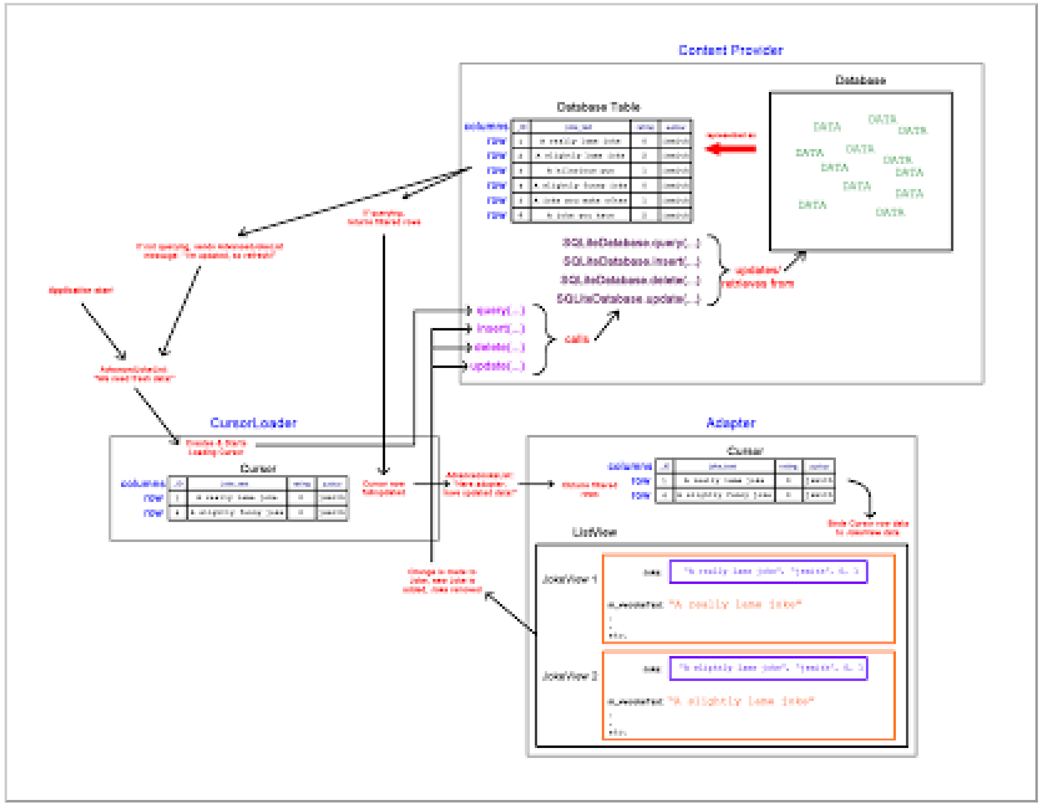
- Back in **AdvancedJokeList**, update **filterJokeList(...)**:
    - Because of when this method gets called, you need to explicitly set the

new value of **m_nFilter** here.

    - Make a call to **fillData()** to complete the refreshing cycle.

- Update **addJoke(...)**:
    - Create a new URI variable that will tell the content provider's URIMatcher in **insert(...)** that the row with **joke.getID()** needs to be inserted.
    - Create a new ContentValues variable and use **ContentValues.put(...)** to place the joke text, rating and author inside of it.
    - Call **getContentResolver().insert(...)**, passing in the URI and ContentValues variables.
        - Make sure to set the Joke's ID equal to the return value from this insertion call! Remember that **JokeContentProvider.insert(...)** returns the ID of the newly inserted row.
    - Make a call to **fillData()** to complete the refreshing cycle.

For your convenience, here is an overview of how data in the application gets passed around (click image for full size):



Run your application. It should behave in nearly exactly the same way as it did in Lab 3, only now all data should be persistent upon creation, destruction, and resumption of the application. The one difference is that jokes that get added are now automatically filtered (e.g., if adding a new joke and the current filter is set to Like or Dislike, then the new joke will be added but automatically filtered out).

If it works as indicated above, then congratulations! You got through a particularly nasty, confusing and dense concept. Hopefully this was an informative exercise in data persistence.

6. Deliverables

To complete this lab you will be required to:

1. Put your entire project directory into a .zip or .tar file, similar to the stub you were given. Submit the archive to PolyLearn. This effectively provides time-stamped evidence that you submitted the lab on time should there be any discrepancy later on in the quarter. The name of your archive should be **lab4<userid>** with a **.zip** or **.tar** extension. So if your username is **jsmith** and you created a zip file, then your file would be named **lab4jsmith.zip**.
2. Load your app on a mobile device and bring it to class on the due date to demo for full credit.
3. Complete the survey for Lab 4**:** https://www.surveymonkey.com/s/436F13Lab4

Primary Authors: James Reed and Kennedy Owen
Adviser: Dr. David Janzen

Comments

You do not have permission to add comments.