Module: Mobile Application development (Android)

Session 43: Instance State, SharedPreferences, SQLite DB and Cursors - Part 1 (Practice)

### This lab will be a continuation of session 19. You will be using your session 19 solution to begin with this lab. You will be persisting data using an SQLite Database and preserving the state of an application during its lifecycle. 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 an SQLiteDatabase

### How to manage database connections

### How to insert, update, remove, and retrieve data from an SQLite Database

### How to work with and manage Cursors

### How to use CursorAdapters

### Activities

### For this lab we will be extending the "Joke List" application that you created in Lab3. This version of the app will add a persistence layer to the previous version. 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 as well 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.

### 1. Setting Up...

### 1.1 Creating the Project

### To begin, download and extract your lab3<usernmae>.zip file which you have submitted as part of Lab3.

### 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.*

### Next you will need to setup a "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 and edu.calpoly.android.lab4.tests, and that the build target is Android 1.6.

### 1.2 Familiarize Yourself with the Source Code

### The skeleton project that you will be using for this lab contains a fully functional solution to Lab3. In particular, the AdvancedJokeList, JokeListAdapter, Joke, and JokeView classes are fully functioning classes from Lab3. Additionally, completed advanced.xml and joke\_view.xml layout files have been supplied as well.

### It is a good idea, and good practice, to read through the source code. See how it compares to your implementation of Lab3. It is especially important to do this if there were any parts of Lab3 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.

### 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 -1.

### 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. The AdvancedJokeListTest2.java Unit Tests should run here at the beginning, but they will start failing later in the lab. You do not need to modify them to continue passing.

### 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 for 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](http://developer.android.com/reference/android/app/Activity.html#SavingPersistentState).

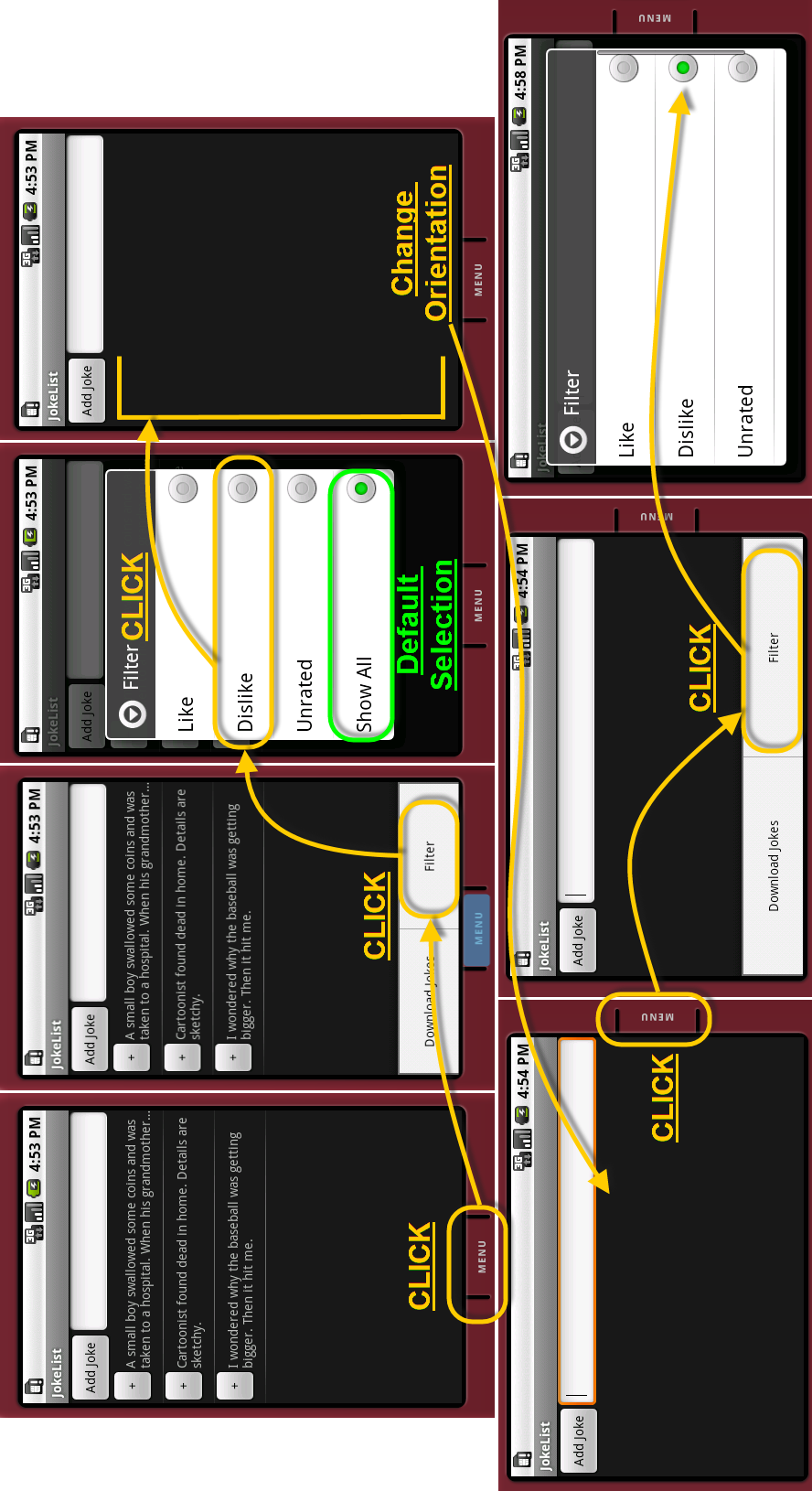
### 2.1 Instance State

### 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.

### It is important to note that this Bundle object might not contain anything if this is the initial creation of the activity. The onRestoreInstanceState 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 these next two sections you will be persisting the Filtering mechanism as instance state. 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](http://developer.android.com/guide/appendix/faq/commontasks.html#appstate) and the Android Documentation on the [Bundle Class](http://developer.android.com/reference/android/os/Bundle.html). When finished, your application should function as depicted by the figure below:



### 2.1.1 Saving Instance Data.

Begin by familiarizing yourself with the Android Documentation on the [Activity.onSaveInstanceState(...)](http://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)) 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. 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.*

### 

### 2.1.3 Restoring Instance Data.

Begin by familiarizing yourself with the Android Documentation on the [Activity.onRestoreInstanceState(...)](http://developer.android.com/reference/android/app/Activity.html#onRestoreInstanceState(android.os.Bundle)) method. It is good to understand what the default implementation does before modifying it.

* Override the **Activity.onRestoreInstanceState(Bundle savedInstanceState)** method.
* 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.

* 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.*

* 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.

The filtering logic is located below the onOptionsItemSelected(...) method, inside of the **setAndUpdateFilter(int newFilterID)** method.

***newFilterID*** *is the MenuItem ID of the new filtering option that should be used. This contains one of the four possible values: R.id.like\_menuitem, R.id.dislike\_menuitem, R.id.unrated\_menuitem, or R.id.show\_all\_menuitem.*

You need to call this method from onRestoreInstanceState(...) passing in m\_nFilter.

* Try running your application:

The default filter should be "Show All" when the application starts up.

Select Menu->Filter

The "Show All" MenuItem should be checked.

Select the "Dislike" MenuItem.

All the jokes should disappear since by default they are unrated at this point.

Change the orientation of your device, which will force your Activity to be destroyed.

No jokes should be displayed since the "Dislike" filter should have been saved, restored, and applied.

*On an emulator you can do this by turning the "Num Lock" off on your keyboard, and then hitting the "7" key on your Number-Keypad. This will put you into Landscape orientation. Hitting the "9" key on your Number-Keypad switches you back to Portrait orientation.*

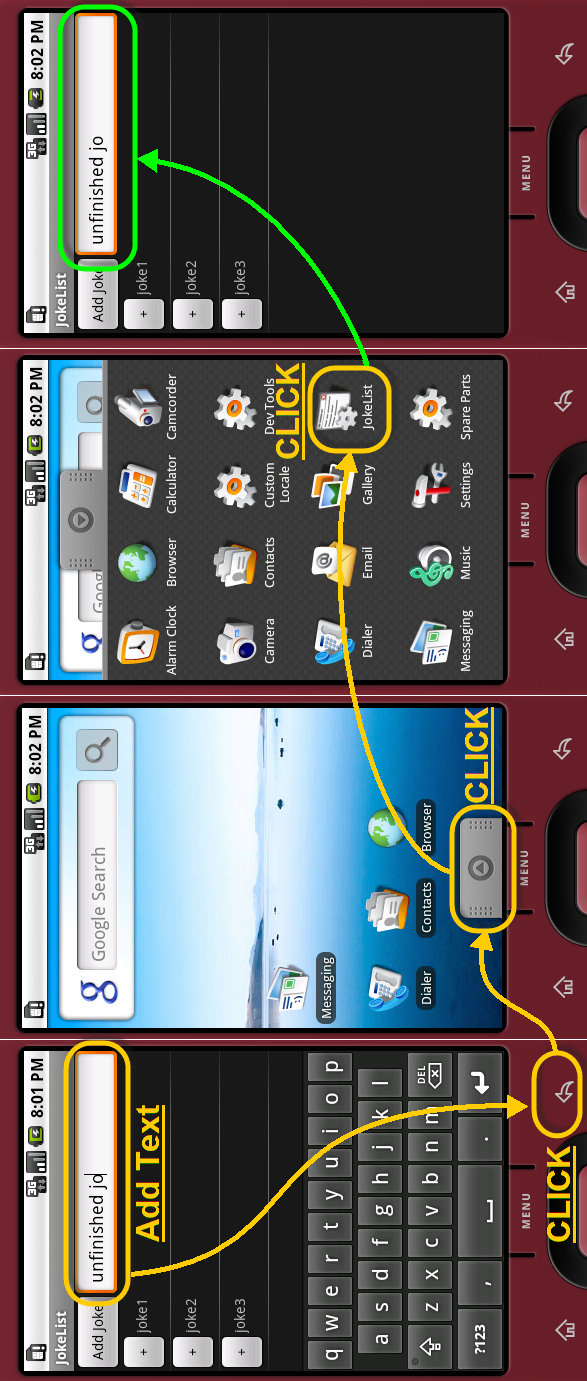
*If you're on a Laptop and don't have a Number-Keypad you may have to play around with your keyboard to get this to work, or you could plug in a USB keyboard from one of the Lab Machines.*

Select Menu->Filter and the "Dislike" MenuItem should still be checked.

### 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](http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle). When finished, your application should function as depicted by the figure below:



### 2.2.1 Saving SharedPreference Data

Begin by reading the Android Developer guide on using [SharedPreferences to save data](http://developer.android.com/guide/topics/data/data-storage.html#pref), as well as the Android Documentation on the [Activity.getPreferences(...)](http://developer.android.com/reference/android/app/Activity.html#getPreferences(int)) method and the [SharedPreferences](http://developer.android.com/reference/android/content/SharedPreferences.html) 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 Documenation on* [*Activity.getPreferences(...)*](http://developer.android.com/reference/android/app/Activity.html#getPreferences(int)) *for details on how to do this.*

* Retrieve a SharedPreferences.Editor from the SharedPrefence object.

*See the Documenation on* [*SharedPreferences.edit()*](http://developer.android.com/reference/android/content/SharedPreferences.html#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 the **AdvancedJokeList.SAVED\_EDIT\_TEXT** as the key.

* Don't forget to commit your changes.

### 2.2.2 Restoring SharedPreference Data

The following should be done in the onCreate() method.

* 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 for details on how to test this.

### 3. Binding Views to a Database

### Your ultimate goal at the end of this lab is to be able to persist Jokes in a Database. In section 4 of this lab, you will be implementing the Database. Before you do that, however, it seems more pertinent to update your application to be able to use a database first. Proceeding in this manner will allow you to incrementally add Database functionality and then test it in section 4. Since the Database implementation will be more tricky, this seems like a better route than the alternative of implementing the Database in one-shot and incrementally updating your application to use it.

### With that said, it may seem a little odd that you will be making use of Database related classes and methods that you either haven't implemented or haven't learned about yet. Fear not, you will learn about and implement the Database related stuff in the next section. For the time being, this section of the lab will act like these features are already implemented (i.e. there will be lots of hand-waving and urging you not to ask questions about the strange-man in the corner of the room).

### In this section of the Lab you will be updating your application to bind the list of Jokes that are displayed, to the results of a database query, as well as write any new Jokes and changes to existing Jokes back to the Database. In particular, you will be creating a call back mechanism in the JokeView class which will notify the AdvancedJokeList Activity when the internal state of a Joke has changed and needs to be written back to the Database. You will then implement the JokeCursorAdapter. This class extends the CursorAdapter class and is very similar to the Adapter that you wrote in Lab 3. The main difference being that JokeCursorAdapter is bound to a Cursor object, which represents the results of a Database query. You will then update the AdvancedJokeList Activity class to use this new Adapter instead of the JokeListAdapter class that you wrote in Lab 3. Lastly, you will update the AdvancedJokeList to add Jokes to the Database instead of to an ArrayList<Joke> and to monitor and write back Joke ratings to the database. The last part will allow you to persist a list of Jokes across multiple application sessions.

### 3.1 JokeView.OnJokeChangeListener

### *I apologize in advance, this is a bit long winded, but it's a complicated subject. Especially for those who aren't well practiced with the Model-View-Controller, Observer/Observable, and Data Persistance patterns. If you already know this stuff, then feel free to skip this introduction.*

### When the list of Jokes is displayed, Jokes are copied from the Database into Joke Objects, which are then wrapped in JokeViews. These JokeViews allow the user to 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 rating gets preserved we must be able to write that rating change back to the 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? 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, the JokeView's job is to show what a Joke looks like and provide controls for manipulating the 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.

### So 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. We could just add an arbitrary method to AdvancedJokeList and call it from the JokeView, however, this is a bit hacky. It is also foreseeable that we might even have a third highly specialized class whose only responsibility is persisting the state of Jokes, like a [Data Transfer Object (DTO)](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FData_Transfer_Object&sa=D&sntz=1&usg=AFrqEzcVy94TmWmKYETxIpA5Uewjt3lFwQ). With this in mind, its easy to see that the JokeView shouldn't even care who handles persistence or how they handle it since its not the JokeView's job to worry about such things. Why should the JokeView cater to the needs of arbitrary classes?

### Instead, the JokeView is just going to put its foot down and say:

### *"If you want to know when the Joke changes, then you've got to tell me you're interested in such things. Furthermore, when the joke changes I'm gonna tell you 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.

### 3.1.1 Registering & Notifying OnJokeChangeListeners

### The JokeView class now contains a member variable reference to a single OnJokeChangeListener, m\_onJokeChangeListener. It should notify m\_onJokeChangeListener any time the internal state of its Joke object changes. You must now maintain this reference and make sure it gets notified:

### In the JokeView class, initialize the m\_onJokeChangeListener to null in the constructor.

### *It is perfectly acceptable for this m\_onJokeChangeListener to be null. No one might care if the Joke changes. You should keep this in mind and test for null when working with this reference.*

### Fill in the setOnJokeChangeListener(...) method.

### Fill in the notifyOnJokeChangeListener() method.

### Notify the m\_onJokeChangeListener that the JokeView's Joke has changed.

### *There is only one way to do this. If you are confused, read the comments in the JokeView.OnJokeChangeListener interface.*

### Call notifyOnJokeChangeListener() whenever the internal state of the JokeView's Joke object changes.

### *This should happen when any of the Joke.set... mutator methods are called (i.e. Joke.setAuthor(...))*

### *Currently this only happens in one method, when the rating is changed.*

### 

### 3.2 JokeCursorAdapter

### The JokeCursorAdapter class will be responsible for providing JokeViews for Jokes in the same way that your JokeListAdapter did in Lab3. The key difference here is that the JokeCursorAdapter will use a Cursor object as its data source instead of an ArrayList<Joke>. A Cursor object is a set of results from a Database query. If you've ever worked with a Database, you can think of the Cursor as an iterator for a list of rows returned from a database query. You can move the cursor around in the list and retrieve data from a particular row. Each row contains all the data for a single joke, the text of the joke, the author, the rating and a unique id.

### I strongly encourage you to read the sections of the Android Developer Guide on the [CursorAdapter Class](http://developer.android.com/reference/android/widget/CursorAdapter.html) before beginning.

### 3.2.1 Fill in Basic JokeCursorAdapter Methods

* Begin by filling in the constructor.
  + Set m\_nSelectedID to Adapter.NO\_SELECTION.
  + Set m\_listener to null.
* Fill in the **getSelectedID()** method.
  + Instead of maintaining the position in a list that is "selected" like the JokeListAdapter, the JokeCursorAdapter will maintain the ID of the Joke that is "selected".
  + This method replaces JokeListAdapter.getSelectedPosition().
* Fill in the **onItemLongClick(...)** method.
  + Set m\_nSelectedID equal to the id argument.
* Fill in the **setOnJokeChangeListener(...)** method.
  + Set m\_listener.
  + This onJokeChangeListener will be applied to every JokeView that this adapter creates/recycles.

### 3.2.2 Make JokeCursorAdapter Extend CursorAdapter

### Make JokeCursorAdapter explicitly extend the android.widget.CursorAdapter class

### Uncomment the line making a call to super(context, jokeCursor) in the constructor.

### Add the necessary abstract methods:

### public void bindView(View view, Context context, Cursor cursor)

### public View newView(Context context, Cursor cursor, ViewGroup parent)

### *Instead of having one method that recycles old View objects and creates new View objects like the JokeListAdapter.getView(...) method class, the CursorAdapter separates this logic out into the two separate methods listed above.*

### Fill in bindView(...).

### This method recycles JokeViews that have been previously created by this Adapter and are no longer in use.

### Retrieve the Joke from the Cursor parameter.

### Make a call to the static JokeDBAdapter.getJokeFromCursor(cursor) method.

### *This method (which you haven't implemented yet) parses the data from a database row that the Cursor object is pointing at, constructs a Joke object from the data, and returns it.*

### Set the recycled view parameter's Joke to the Joke object you just retrieved from the Cursor.

### *Hint: You can safely cast view to a JokeView object.*

### Set the recycled view parameter's onJokeChangeListener to m\_listener.

### Fill in newView(...).

### This method creates new JokeViews.

### Retrieve the Joke from the Cursor parameter like you did in bindView(...).

### Set the JokeView's Joke to the Joke object you just retrieved from the Cursor.

### Set the JokeView's onJokeChangeListener to m\_listener.

### Return the JokeView.

### 3.3 Updating AdvancedJokeList

### For this section you will be updating AdvancedJokeList to use the the JokeDBAdapter Database class and JokeCursorAdapter class that you just implemented. You will start by initializing your database connection and updating the types on both m\_jokeAdapter and m\_arrJokeList to JokeCursorAdapter and Cursor respectively.

### AdvancedJokeList will query the database for a list of Jokes to display, which will return a Cursor. This Cursor is what you will use in place of the ArrayList<Joke> that you used in Lab 3. Its worth noting that this will make filtering much easier, since any time a filter is changed you simple have to re-query the database and get a new Cursor. Consequently, you will have to update your filtering functionality.

### Lastly, you will be updating all of the JokeList maintenance methods. Specifically, you will update the addJoke(...) method, the Remove Joke Context-MenuItem, the Upload Joke Context-MenuItem, and make AdvancedJokeList save changes to Jokes in the database. You will do this last part by making AdvancedJokeList implement the onJokeChangeListener interface so that it can monitor and write back Joke changes to the database.

### 3.3.1 Initializing JokeDBAdapter, JokeCursorAdapter, & Cursor

### Change the type on m\_arrJokeList from ArrayList<Joke> to Cursor

### Change the type on m\_jokeAdapter from JokeListAdapter to JokeCursorAdapter.

### In the onCreate(...) method:

### Initialize m\_jokeDB.

### *m\_jokeDB is a reference to a JokeDBAdapter object, which represents the connection to your Database. Any interaction with the Database will be done through m\_jokeDB.*

### Set m\_jokeDB equal to a new JokeDBAdapter object. The constructor takes a reference to an application Context object, which is *this*.

### Open the connection to the Database by calling m\_jokeDB.open().

### Since the Filter is set to "SHOW\_ALL" by default when the activity starts up, you will need to retrieve a Cursor pointing to a list of all the jokes in the Database.

### Retrieve a Cursor object for all Jokes in the Database by calling m\_jokeDB.getAllJokes().

### Set m\_arrJokeList equal to the Cursor you just retrieved.

### A Cursor object represents an open connection to the Database, and has a lifecycle just like an Activity does. When the Activity closes, we want to make sure that all of our Database connections close as well. Luckily, the Activity class provides a mechanism for managing the lifecycle of a Cursor so that it falls in line with the lifecycle of the Activity.

### Have AdvancedJokeList manage the lifecycle of m\_arrJokeList by calling startManagingCursor(...) and pass in m\_arrJokeList.

### Initialize m\_jokeAdapter to a new JokeCursorAdapter.

### Make sure that m\_jokeAdapter binds to m\_arrJokeList by passing m\_arrJokeList into the constructor.

### Remove the code that initializes your list of Jokes from the joke-string resource values. Since jokes will be persisted in the database you no longer need to prime you application with test jokes.

### 3.3.2 Update Your Filtering Functionality

### Since the master copy of all your jokes will now live in the Database, and m\_arrJokeList is now a copy of that list, there is no need for an extra mechanism to keep track of which jokes should be displayed to the user. You need to remove the extra mechanism being used to filter Jokes so that all the jokes in m\_arrJokeList will get displayed to the user. When you need to change the filter, you simply retrieve a different Cursor from the Database and update m\_arrJokeList to reference the new cursor.

### In this section you will be updating the filtering logic to retrieve a new Cursor from the database instead of manually filtering the Jokes. The filtering logic will be self-contained in a single method called AdvancedJokeList.setAndUpdateFilter(...)*.* Perform the following modifications in that method:

### Since we are getting a new Cursor from the Database, tell AdvancedJokeList to stop managing the lifecycle of our current Cursor and close it.

### Make a call stopManagingCursor(...), passing in m\_arrJokeList.

### call close() on m\_arrJokeList.

### Retrieve a new Cursor from m\_jokeDB, update m\_arrJokeList to use it, and make sure AdvancedJokeList manages its lifecycle.

### call m\_jokeDB.getAllJokes(...), passing in the new filter value as a string, or null if the filter is SHOW\_ALL.

### You will have to update your logic to choose the correct new filter value as a string.

### *The overriden JokeDBAdapter.getAllJokes(String filterVal) method takes a string containing the filter value and returns a Cursor of all jokes that have a rating equal to the filter value.*

### *The filterVal string passed in should contain one of the Joke classes "static-final" rating values. Either Joke.LIKE, Joke.DISLIKE, or Joke.UNRATED. Alternatively if you set filterVal to null then the method will not filter out any jokes and will return a Cursor of every Joke in the Database.*

### Set m\_arrJokeList equal to the Cursor you just retrieved.

### Make a call to startManagingCursor(...), passing in m\_arrJokeList.

### Since we have changed the Cursor that we are using, we now need to update m\_jokeAdapter to use this new cursor as well.

### Call m\_jokeAdapter.changeCursor(...) passing in the new Cursors.

### *Be aware that calling changeCursor(...) will cause the Adapter to close whatever Cursor the Adapter was previously using. You were asked to explicitly close() the cursor yourself to get practice doing that.*

### 3.3.3 Maintaining the JokeList

### Update the addJoke(...) method so that it inserts the Joke parameter passed into the Database and have the Cursor refresh itself.

### Instead of adding the new Joke into m\_arrJokeList, call m\_jokeDB.insertJoke(...) passing in the new Joke.

### Call requery() on m\_arrJokeList to have the Cursor refresh itself.

### Update the Remove Joke Context-MenuItem so that the Joke is properly removed from the Database and have the Cursor refresh itself. These changes should be made to the Remove MenuItem's OnMenuItemClickListener.

### Retrieve the ID of the selected joke from m\_jokeAdapter by calling its getSelectedID() method.

### pass the ID of the Joke to m\_jokeDB.removeJoke(...).

### Requery m\_arrJokeList.

### Update the Upload Joke Context-MenuItem to get the Joke from m\_jokeDB instead of m\_arrJokeList. These changes should be made to the Upload MenuItem's OnMenuItemClickListener.

### Get the ID of the selected joke from m\_jokeAdapter.

### Retrieve the Joke from the Database passing the Joke ID into m\_jokeDB.getJoke(...).

### Upload the Joke as you did before.

### Make AdvancedJokeList monitor changes to any Joke and save them back into the database. Do this by making AdvancedJokeList implement the JokeView.OnJokeChangeListener interface and setting the m\_jokeAdapter's OnJokeChangeListener to AdvancedJokeList.

### Implement the JokeView.OnJokeChangeListener interface, then create and fill in the required onJokeChanged(JokeView view, Joke joke) method:

### This method takes as a parameter the JokeView that contains the Joke that is being changed, as well as the Joke object that was changed.

### Pass the Joke object that was changed into m\_jokeDB.updateJoke(...).

### Requery m\_arrJokeList.

### Set m\_jokeAdapter's OnJokeChangeListener to reference this instance of AdvancedJokeList Activity by calling its setOnJokeChangeListener(...) method.

### This should be done in the AdvancedJokeList.onCreate(...) method, just below the initialization of m\_jokeAdapter.

### And you are done updating your JokeList Application to persist Jokes to a Database. Now all thats left is to implement your JokeDBAdapter Database class. Before we go on, I encourage you to test your application using the JokeCursorAdapterTest provided. This set of unit tests uses a provided database implementation (the same as what you will be completing in the next section). To use it, you will need to import the JokeDBAdapter (import edu.calpoly.android.lab4\_key.JokeDBAdapter;) from the key in your AdvancedJokeList and JokeCursorAdapter classes. After testing, you can remove these imports so you use your own versions of these classes.

### 