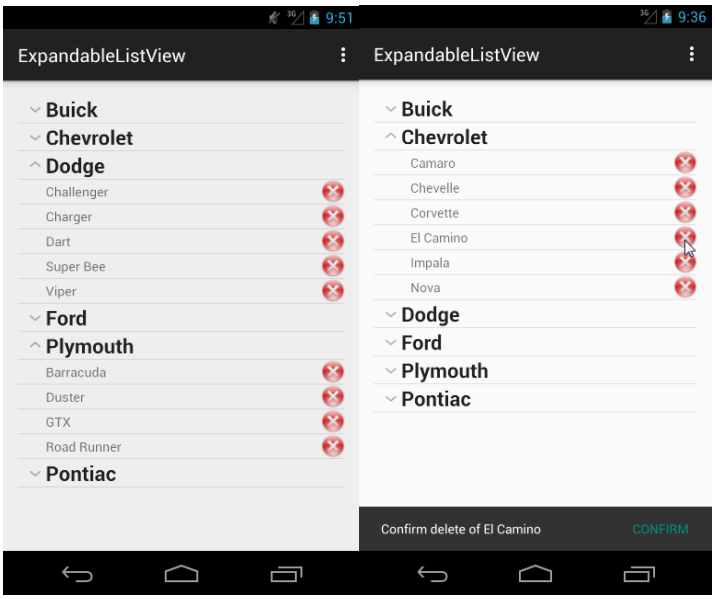
CSCD 372: Android Development
Lab 3: Custom Adapters and Expandable List Views

Lab Overview

The purpose of this lab is for you to gain some familiarity with custom Adapters by providing one for an Expandable List View. While it is possible to create list views and expandable list views from static data contained in XML files, we'll be doing it programmatically so that you are prepared to handle dynamic data. You'll also learn how to use Tags attached to Views and access embedded Assets. In this case the asset will be a text file that you will parse for content to populate the Expandable List View. That content will be the make (manufacturer) and model of various muscle cars. The manufacturers constitute what the list refers to as "group" items, and the models are "child" items. The user will be allowed to remove an entry by pressing the X (because no one really wants to see an El Camino or a Duster on a list of muscle cars). Feel free to use your own two-level data set for this.



You need not follow this order of development, but I recommend that you at least glance through the steps because they provide some additional details on the required behavior.

Task 1: Create an empty project shell and modify the main activity layout

1. Start with the blank or empty template and drop an expandable list view widget onto your main layout.

Task 2: Add layout resources for the view of a group item and the view of a child item

1. You can add a template layout resource by right-clicking on your layout folder and selecting New→Layout resource file. Do that to create a layout for the view of each group item. Base the layout on LinearLayout, and give it a Large TextView widget. Set the text style to Bold. Note that you are only creating a layout for the view of a single group item. The ExpandableListView will handle replicating that.
2. Find a .png or .jpg icon online to represent the delete image. You can pick your favorite, as long as it clearly represents deletion, like an X or a trash can or something. Paste it into your drawable resource

directory. You'll be specifying it as the source for an ImageView widget, and can fine-tune the scaling on that widget.

3. Create another layout for the child item view. For this you can use either RelativeLayout or LinearLayout (horizontal). Arrange a TextView widget on the left side, and an ImageView on the right side. Set the source image property for your ImageView. No, you do NOT need an ImageButton, even though we will be handling onClick events. The button just adds unnecessary bling, and size, in this case. Using an ImageButton will also cause a problem for you in Task 8 below (for an extra challenge, see if you can find a way to make it work with an ImageButton). If you use a RelativeLayout, look for the necessary alignment properties. If you use a LinearLayout, you can force the ImageView to the right by giving the TextView a layout:weight of 1 along with a 0 dp width. Also give the TextView a layout:gravity of center_vertical so that it aligns vertically with the center of the LinearLayout (and hence the center of the ImageView, which will be taller). The LinearLayout should, of course, have a layout:height of wrap_content.

Task 3: Add a Class to represent a Manufacturer

1. Right-click on your src directory and add a new Java class that represents a manufacturer. Give it a string member variable for the name of the manufacturer and another member variable that is some kind of collection of Strings for the muscle car models produced by this manufacturer. I used an ArrayList for that.
2. In addition to a constructor, you'll likely want methods for getting the manufacturer name, getting a particular model name (by position in the collection), deleting a particular model, getting the number of models, and adding a model (or maybe adding an array or list of models).

Task 4: Add an Adapter that extends BaseExpandableListAdapter

1. Add an Adapter class that extends BaseExpandableListAdapter.
2. Give it some member variables of type Activity (to hold a reference to the creating activity) and some kind of a collection of manufacturer objects (I again used ArrayList here).
3. Given the preceding, your adapter constructor should have a signature something like this:

```
public MyListAdapter(Activity act, ArrayList<Manufacturer> manufacturers)
```

Task 5: Add necessary overrides to your Adapter

There are ten methods that provide the ExpandableListView with the information it might need from the adapter. They are (get the method signatures from the API):

1. getChild(): Return the child object (model string) from your collection, using the passed group and child position indices.
2. getChildId(): Return a unique identifier for the given child in the given group. In this case you can just return the child position as an identifier.
3. getChildView(): Here you need to provide a View for a child at the given position. If the passed convertView is null, inflate a child layout (which first requires obtaining a LayoutInflater from the Activity). Then set the TextView widget on the inflated layout to the correct model string, using the passed group and child positions. We'll worry about handling onClick for the delete image later, so do not create a handler for that yet. Return the inflated view.
4. getChildrenCount(): Return the size of the (inner) list of models for the passed group.

5. getGroup(): Return a group object from your list of manufacturers, using the passed group index.
6. getGroupCount(): Return the size of your list of manufacturers.
7. getGroupId(): Return a unique identifier for the given group. In this case you can just return the group position as an identifier.
8. getGroupView(): Here you need to provide a View for a group at the given position. If the passed convertView is null, inflate a group layout. Then set the TextView widget for the manufacturer to the correct string obtained from your list of manufacturers. Return the inflated view.
9. hasStableIds(): The expandable list view calls this to determine whether your IDs remain the same when the list data changes. We're going to lie here and say that they do, (by returning true) because it won't matter in this example that the ID for a child might change as a result of deleting some other child. We won't need a unique ID when we respond to a click on a child, and we won't need the list to re-query the IDs for various positions when we delete a child (which it will do if we return false).
10. isChildSelectable(): The list wants to know if a particular child is selectable. The answer for any such child is true, as we will be taking an action for such selections.

Task 6: Create and parse the text file

1. In your Activity class, add a collection to hold your manufacturer objects. It should be the same type of collection as you used in your adapter, since you will be passing that in from the Activity. Don't forget to initialize it by calling the collection constructor.
2. Create an Assets folder. There are a couple of ways to do that. In the Android view of your project it should appear in the app directory at the same level as your java and res folder. If it doesn't, something is wrong. Place a text file in that folder that contains the following makes and models. I highly recommend that you keep this format as it will make a possible later lab easier for you:

   Buick,GTX,Gran Sport
   Chevrolet,Camaro,Chevelle,Corvette,El Camino,Impala,Nova
   Dodge,Challenger,Charger,Dart,Super Bee,Viper
   Ford,Galaxy,GT,Mercury Cougar,Mustang,Thunderbird
   Plymouth,Barracuda,Duster,GTX,Road Runner
   Pontiac,GTO,Olds 442
3. Add a private method called something like parseFile() that takes the filename as a String. Have it return a boolean to indicate whether the parse was successful or not. In that method make a call to getResources().getAssets() to get an instance of an AssetManager. On the AssetManager, call open(fname) to get an InputStream on the file. You'll need to catch an exception here – just return false if it happens.
4. Parse the file and populate your two lists. You can do this however you like. I won't be testing your solution for how it responds to incorrectly formatted files, because as an embedded asset the content is completely under your control, but use reasonably robust file parsing practices nevertheless. For the format above, I found it convenient to attach a Scanner to the file and just read one line at a time into a String. I then used String.split() to separate the line into more Strings.

Task 7: Launch the list and test

1. In your Activity's onCreate() method, call your method that parses the file.
2. If that that fails, put up a Toast that says so. If it is successful ...

3. Create an instance of your adapter class, passing it your collection. Get a reference to your ExpandableListView and call setAdapter() on it, passing it your adapter.
4. Do a test run to see if you get the expandable list you were expecting. Your delete icons should appear on child views, but there is not yet any reaction to clicking on the child or on its delete icon.
5. You might end up with your group names on top of the list expander doohickey. Fix that by modifying your group layout. Retest until it looks decent.

Task 8: Handle user selection of a model

1. Modify your Activity to implement ExpandableListView.OnChildClickListener, and make the call to register as a listener with the ExpandableListView.
2. Add the required onChildClick() method. The @Override annotation was extended to interface methods as of Java 6, and you really should be using it to make sure your signatures are correct.
3. Use the passed group and child positions to put up a Toast containing the manufacturer and model names. Obviously it'd be more interesting to pop up a picture or something. Feel free if you're ambitious.
4. Do a test run to make sure it works.

Task 9: Handle the delete action

1. Go back to your adapter class. In the getChildView() method, register an OnClickListener with the ImageView holding the delete icon. The easiest approach here would be to create an anonymous inner class for the listener right here with the registration call. In this case I want you to take a slightly more difficult approach, for two reasons. First, it will illustrates how anonymous inner listeners are advantageous in this case. More importantly, it allows me to introduce another Android technique that becomes useful in other circumstances. So please register “this” as the listener.
2. That means you need to make your Adapter class implement View.OnClickListener, and give it an onClick() method (which receives the View being clicked). What we need to do in that method is delete the corresponding entry from our list of lists ...
3. Hmmm, we have a problem. In onClick(), the View we receive tells us nothing about which child entry (car model) needs to be deleted – it’s just the View for the delete image. Note that we would have this information if our listener was implemented as an anonymous inner class inside getChildView. On the other hand, we would have been creating a unique instance of the handler for every child that is currently expanded. You can argue whether that is sensical amongst yourselves. Here we do something different. When we create each child view, we can attach one or more “tags” to it, such that the View carries any additional information we want ...
4. In getChildView() make two calls to setTag() on the delete ImageView object. One to store the group position, and another to store the child position. When you use more than one tag you must supply identifiers for them. Android wants to ensure that the tag names you use are unique, and so insists that they come from resource identifiers. So ...
5. Right-click on your res/values directory and select New→Values resource file. The filename isn’t particularly important here, but it would be meaningful to call it tags(.xml). Create a couple of tag ids in that file as follows (inside the <resources> brackets):

```
<item type="id" name="group_num" />
<item type="id" name="posn_num" />
```

You can now use those tag names in your setTag() calls (they are resources, so retrieve them in the same way you retrieve any resource id).

6. Back to onClick(): call getTag() on the passed View object twice to get the group and position numbers. Use those to remove the model entry from your collection. Then make a call to notifyDataSetChanged() to let the adapter (and view) know that the list being displayed has changed.
7. Run and test. Everything should now be working as planned. Almost ...

Task 10: Rotations and Delete Confirmation

1. Try deleting a few car models and then clicking a model that was below the deletion point. Why is your Activity able to display the model from positions sent by the adapter, when the adapter's copy of the collection has been modified from what it was initially given? If the answer is obvious to you, good, and you then know how to fix the following problem ...
2. Run your app, expand a group, delete a couple of models, and then rotate your display to landscape and back. Did the deleted items come back? Fix it so they don't, and NOT by disabling landscape mode in the manifest. Hint: make your collection Serializable, which is something you can do with two words worth of code.
3. One last thing. Let's be polite and ask the user to confirm a deletion. Do that by popping up a Snackbar in the delete handler, as shown above. Make a call to setAction() on the Snackbar in order to get the Confirm button, and move your deletion code to that action handler. I used an anonymous inner class for this handler. Note that if we had done all this inside getChildView(), then we would have had an anonymous inner handler inside an anonymous inner handler. I'm not saying that would be inappropriate (it would let us forgo tagging here), but I do think the resulting code for that would be ugly.
4. Make sure you have an "About Box" and submit your apk file using the required naming convention. Please also submit the java source for your main activity and adapter classes with this lab.