CS 370 Lab 5: RecyclerView

This lab is about displaying a list of items, in a memory-efficient manner.

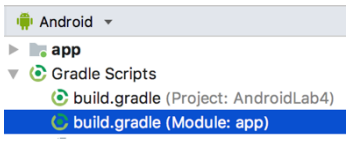
Obtain Code

1. Ensure that the lab workstation is booted into OS X
2. Create a new folder on your desktop called Repositories
3. Open a terminal session (Applications -> Utilities -> Terminal)
    - This is just a terminal window on your local machine! ***Do not log in to blue!***
4. Ensure there are no other default accounts in the OSX keychain (if on public/lab computer):
    a. Keychain management instructions: https://kb.wisc.edu/helpdesk/page.php?id=2197
    b. Search for any *github.com* entries and remove them
5. Using the command line, change directory to the Repositories folder you just created
6. Clone the repository: **git clone https://github.com/ssu-cs370-s19/AndroidLab5.git**
7. Change directory to the AndroidLab5 folder you just cloned
8. Create a new branch in the git repository called **LastnameFirstname**
9. Open the lab in Android Studio

Part 1: Parsing with Gson

Gson is Google's JSON parsing library. If your Java model structure matches the JSON you are parsing, Gson can automatically create your models and fill them with data.

1) Look in the Gradle Scripts directory, open the app build.gradle file



2) The dependencies { } block declares all the libraries our app needs to use. Uncomment the *gson* dependency, and rebuild the project. This will download the Gson files from the gradle server, and make it available for use in the app.

To use Gson, we need to make sure our Java models match the response structure.

3) Look in the model directory. You should see both *RecipeModel* and *RecipeResponse*.
4) Open *RecipeModel*. Compare the variables to the fields in the JSON sample below:

```
{
 "recipeName" :  "Homemade Blue Lemonade"
 "id" :  "Homemade-Blue-Lemonade-2536078"
 "rating" :  4
 "smallImageUrls":  [  "example.com"  ]
}
```

```
class RecipeModel {
  private String recipeName;
  private String id;
  private int rating;
  private List<String> smallImageUrls;
}
```

Gson will parse the entire response for us, **if** the Java object hierarchy lines up with the JSON. Since the response itself is one big JSON object, we need a corresponding Java class.

5) Open *RecipeResponse*, and compare against the JSON sample.

```
{
 "totalMatchCount":  12345
 "matches":  [  …  ]
}
```

```
class RecipeResponse {
  private int totalMatchCount;
  private List<RecipeModel> recipes;
}
```

matches contains a list of objects. Those objects are already modeled in Java with the RecipeModel class, so we use a List of RecipeModels.

Notice that the JSON key-name is not the same as the Java variable name…

6) Add the *SerializedName* annotation to the *recipes* variable.

```
@SerializedName("matches")
private List<RecipeModel> recipes;
```

This annotation signals to Gson that the variable has a different Java name than what can be found in JSON. Without this, it would search for a *recipes* field in the JSON (and fail).

Now that the models are ready, we can use Gson to fill them with data.

7) Open *RecipeParser*, add this code in *getMatches* to parse the response:

```
// create an instance of Gson
Gson gson = new Gson();

// use Gson to inflate a RecipeResponse
RecipeResponse response = gson.fromJson(json, RecipeResponse.class);
```

8) Get the list of recipes from the RecipeResponse, and return it.

If you run the app, it should look the same as your completed Lab 4 app. Commit your code.

Part 2: Recycling a List with RecyclerView

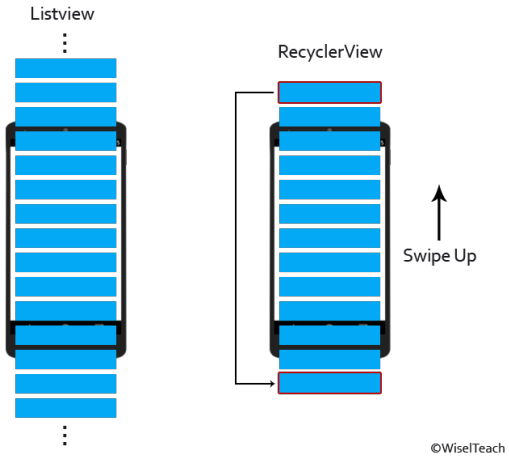
When you have more objects than will fit on the screen, you need a scrollable list.

When you have a lot of objects in your list, you can run out of memory fairly quickly.

A RecyclerView lets you avoid using memory on objects that aren't being displayed.

Each entry in a list is made of two pieces:
- a View (layout / drawing on the screen)
- the data that the View is displaying.

A simple ListView (left) would create a new View for each piece of data it needs to display. As you scroll, it will continue creating more Views for your data, and deconstructing the old Views.



However, if the Views are all the *same layout* and the only difference is the data they are showing, there is no need to delete the View and create a new one. Instead, we can reuse the Views that are no longer on the screen by simply changing which data the View is showing. (The list of actual data is never deleted/rearranged/changed in any way by the RecyclerView.)

There are three parts to a simple RecyclerView:

- The RecyclerView

  The overall container for the list. Empty to start with, and will fill itself with Views full of data that you provide. Uses a LayoutManager to determine how to organize its elements.

- The ViewHolder

  Each ViewHolder is in charge of displaying a single item with a view. The RecyclerView uses only enough ViewHolders as needed to display the on-screen portion of the content, plus a few extra. As the list scrolls, the ViewHolders that move out of vision get rebinded with new data and get brought back onto the screen.

- The Adapter

  The Adapter is the bridge between the RecyclerView and the ViewHolders. It holds the connection to the list of data, and tells the ViewHolders which one to display.
  (screen scrolls to x position → get data from dataList[x] and bind to a ViewHolder)

The RecyclerView starter code is in a different branch, to allow you to test Gson separately first. Make sure you are on your branch (not master), and that your Gson work is committed.

9) Merge the starter code into your branch by running this command in the terminal:

```
git merge recyclerview
```

There shouldn't be any conflicts, but if there are you can try to fix them or ask for help.

Git will ask for a commit message describing the merge. The default one is fine – save and close the editor window, and the merge will be complete.

10) Open the *activity_main* layout. The LinearLayout with two TextViews is gone, and has been replaced by a RecyclerView.
Nothing should go inside the RecyclerView layout here; it will automatically fill itself up once connected to an Adapter (and data) later.

11) Open *res/layout/recycler_list_item.xml*. This is the layout that we will use for each entry in the list. There's two TextViews here: one for a recipe name and one for the rating.
- Notice that the root layout is using *wrap_content* for height – this layout will be used multiple times on one screen, so it needs to not take up all the vertical space.

12) Open *RecipeViewHolder*. This ViewHolder will represent one model from the Data list.

13) In the constructor, after calling *super()*, assign the TextViews to their class variables.
- This class is *not* a subclass of Activity, so *findViewById* won't work here.
- The argument *itemView* is a subclass of View, and it can use the *findViewById* function:

```
itemView.findViewById(…);  // search for an id in itemView's child elements
```

14) The *bindView* method will get called whenever we attach Data to this ViewHolder.
The *model* parameter represents the data that this ViewModel needs to display.

a) set the *itemNameTextView* to display the recipe name from the *model*

b) set the *itemRatingTextView* to display the rating from the *model*

```
// use this to place an integer into a string
String.format("Rating: %d / 5", variable)  // (int)variable replaces %d
```

15) Open *RecipeViewAdapter*. This class oversees creating and assigning data to ViewHolders.
    - Notice the class variable storing a List<RecipeModel> – this is all the data for the entire list. The RecyclerView only creates enough ViewHolders for what can be seen on screen, and will reuse them with different data as the user scrolls through the list.

16) Look at *onCreateViewHolder*. The RecyclerView calls this method of the Adapter when it needs a new ViewHolder. Here we take the *recycler_list_recipe_item* layout and inflate it (create objects from the layout), then create a ViewHolder using the inflated views.

17) The *onBindViewHolder* method is called when we need to change the data in a ViewHolder.
    a) The *recipeViewHolder* argument is the ViewHolder whose data needs to be changed.
    b) The *position* argument tells us where in the list to get new data.
    a) Get the model at the specified position in the list of data
    b) Call *RecipeViewHolder.bindView* to bind the model into the ViewHolder's existing views

```
RecipeModel model = recipeCollection.get(position);
recipeViewHolder.bindView(model);
```

18) The *getItemCount* method allows the RecyclerView to know when to stop scrolling.
    Finish this method by returning the number of items in the *recipeCollection*
    a) in Java, use *.size()* to get the length of a list

The last thing we need to do is set up the RecyclerView and connect the adapter to it.

19) Open *MainActivity*, and look for these two lines of code:

```
RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getBaseContext());
recyclerView.setLayoutManager(layoutManager);
```

    These lines create a new LinearLayoutManager that will simply order items top-to-bottom. Other options include a GridLayoutManager, or creating your own custom LayoutManager.

20) In the *RecipeCallbackListener* callback (defined in the button clicklistener), we are no longer updating a single TextView with one model's information – we will fill the RecyclerView with data and it will take care of displaying it for us (using the adapter and view holder).
    a) Create a new *RecipeViewAdapter* with the list of items we get in the callback.
    b) Pass the adapter to the RecyclerView.

```
RecipeViewAdapter adapter = new RecipeViewAdapter(models);
recyclerView.setAdapter(adapter);
```

    Once the RecyclerView has an Adapter connected, it will use the Adapter to:
    a) find out how many items there are, using *getItemCount*
    b) create enough ViewHolders to cover the visible screen, using *onCreateViewHolder*
    c) bind the Data (starting from position 0) to the ViewHolders, using *onBindViewHolder*

21) If your app is running correctly, you should be able to enter a search term, click the button, and see a list of results displayed. If not, revisit the above steps and make sure you've followed all the instructions.

22) When your app runs properly, commit your changes to your branch and push it:

```
git add .
git commit –m "working code complete"
git push origin yourbranchname
```

    Your code branch is now saved and committed to the git repository.

This completes Lab 5.