CS 370 Lab 3: AsyncTask and JSON Deserialization

This lab is about running tasks on a background thread, making basic HTTP calls, and deserializing models from JSON.

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! You **don't** need to log in to blue
4. Ensure there are no other default accounts in the OSX keychain
    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-F18/AndroidLab3.git**
7. Change directory to the AndroidLab3 folder you just cloned
8. Branch the repository using a branch name of **lastnamefirstname-370H3**
9. Open Android Studio, and open Lab 3

Part 1: AsyncTask

1. Open up *activity_search.xml*, and use the Text editor (not Design)
2. Note the elements already present in the layout:
    a. LinearLayout: the root layout for the associated Activity
    b. TextView: an element for displaying text
        i. note this attribute: `tools:text="recipe name"`
    c. Button: an element allowing the user to invoke an action
    d. EditText: an element allowing users to provide text input
    e. ImageView: an element used to display an image

    The `tools:text` affects the text that appears in the Design tab, but ***not*** when the app is run. This makes it easier to see the planned layout without affecting the actual layout.

3. Look in *SearchActivity*. There should already be member variables declared that match the view elements from the *activity_search* layout.
4. Look in the <u>models</u> directory, and find the *RecipeModel* class.
    a. Add two String variables: *recipeDescription* and *recipeImageUrl*.
    b. Make sure to adhere to the principles of OOP (encapsulation)!
5. Look in the <u>network</u> directory, and find the *RecipeSearchAsyncTask* class.
    a. This class lets us use a background thread to execute long-running tasks
    b. Notice the nested interface *RecipeCallbackListener*, which will let us handle the response back on the main thread.
    c. Add a private *RecipeCallbackListener* class variable *listener*, and a setter for it.
6. Look in the <u>utilities</u> directory, and find the *RecipeParser* class.
    a. This will contain the logic to deserialize the response JSON into a RecipeModel
7. Return to *SearchActivity*. Add code to assign instances of the layout elements to the corresponding private class variables (using *findViewById*).
8. Add a *RecipeCallbackListener* private member variable to the *SearchActivity* class.
    a. Use *recipeCallbackListener* for the variable name
9. Add a click handler for the Button (*setOnClickListener* and *new View.OnClickListener*).
10. In the Button click handler's *onClick* method, add this code:

```
recipeCallbackListener = new RecipeCallbackListener() {
    @Override
    public void onRecipeCallback(RecipeModel recipeModel) {
        recipeName.setText(recipeModel.getRecipeName());
    }
};
```

    This creates a concrete implementation of the *RecipeCallbackListener* interface. If you peek back at where the interface is defined, you should see that it only contains one method. The overridden method here has code that sets the text attribute of the *recipeName* TextView to the value contained in the RecipeModel that was passed as the method's argument.

11. Add more code to the click handler, below the implementation of *RecipeCallbackListener* :

```
RecipeSearchAsyncTask task = new RecipeSearchAsyncTask();
task.setRecipeCallbackListener(recipeCallbackListener);
task.execute(searchEditText.getText().toString());
```

    This creates a new *RecipeSearchAsyncTask* for background thread operations, adds our listener to it, and executes the *doInBackground* function.

12. Open the *RecipeSearchAsyncTask* class and add the following code to the *doInBackground* function, above the return statement:

```
String searchParams = params[0];
OkHttpClient client = new OkHttpClient();
HttpUrl.Builder urlBuilder = HttpUrl.parse(baseApiUrl).newBuilder();
urlBuilder.addQueryParameter("_app_key", apiKey);
urlBuilder.addQueryParameter("_app_id", appId);
urlBuilder.addQueryParameter("your_search_parameters", searchParams);
String url = urlBuilder.build().toString();
Request request = new Request.Builder().url(url).build();
Response response = null;

try {
    // response holds server's answer
    response = client.newCall(request).execute();
    if (response != null) {
        return RecipeParser.recipeFromJson(response.body().string());
    }
} catch (IOException e) {
    // do something with exception
}
```

OkHttp is a library designed for making http requests over the internet. *OkHttpClient* is an object from the library that will make the web connection happen.

HttpUrlBuilder is a builder that constructs the URL that we're connecting to.

Request is a model object representing the request we're trying to make.

Response is a model object that encapsulates the remote server's response. It contains metadata about the status of the request (success, failure, blocked, etc.), as well as the data we requested.

If we received a successful response, we then use the *RecipeParser.recipeFromJson* function to deserialize the json String into a RecipeModel, which we return.

13. Lastly, we need to notify the listener when our task completes.
    a. Override the *onPostExecute* method of *AsyncTask*
    b. Add code to call the *listener*'s callback method, and pass it the *RecipeModel*:

    ```
    listener.onRecipeCallback(recipeModel);
    ```

14. Open *AndroidManifest.xml*. Notice this line near the top of the file:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

This line tells the Android OS that our app needs to use the internet. Without this line, the app will crash as soon as we try to execute the request inside *RecipeSearchAsyncTask*.

Part 2: Deserializing JSON

15. Open the *RecipeParser* class and examine the *recipeFromJson* function.
    a. This function is **static**. We can call it directly from the class itself -- we don't need to create an object of the class (using **new**) in order to call this function.
    `RecipeParser.recipeFromJson(…)`
    b. Note that some work is already done – it converts the String argument into a JSONObject, and gets the *matches* JSONArray from it. Finally, it retrieves the first JSONObject from the array.
16. Finish the *recipeFromJson* method by continuing to break down the JSON response and create a *RecipeModel* to store the data.
    a. For now, we're only interested in the first response in the array. Ignore the rest.
    b. Same for the *smallImageUrls* array – only save the first url into the model.

    Look at the block comment below the class to see an example JSON response.
    (witness the power of an IDE! click in comment ~ press alt-enter ~ inject language ~ json)
17. In *RecipeSearchAsyncTask*, override the *onPostExecute* method.
18. Add code that calls the local instance of *RecipeCallbackListener*'s *onRecipeCallback* method, and pass it the response (this sends the *RecipeModel* instance back to the SearchActivity, where the value will be displayed on the app screen).
19. If your app is running correctly, you should be able to enter a search term, click the button, and see the name of your result pop up. If not, revisit the above steps and make sure you've followed all the instructions.
20. When your app runs properly, execute the following commands to commit your changes to your branch:
    a. git add .
    b. git commit -m "working code complete"
    c. git push origin yourbranchname

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

This completes Lab 3.

For bonus points…

Builder Pattern

Practice implementing the Builder pattern for the *RecipeModel* class.

1. Create an inner *Builder* class, inside the *RecipeModel* class

   ```
   public static class Builder {
   ```

2. Create a private *RecipeModel* variable in the builder class, called *instance*
3. Add a constructor to the builder that creates a new *RecipeModel* and assigns it to *instance*
4. Add setter functions to the builder, and mutate the *instance* appropriately
5. Add a *build()* function that returns the *instance*
6. Make the *RecipeModel*'s constructor private and remove the model's setter functions
7. Update the *RecipeParser* method to use your new builder.
8. Commit and push your code. Use "bonus - builder pattern" as your commit message

Image Loading

Use a library to asynchronously load images into your app.

In the lab, we get the url for an image associated with a recipe. In order to get the actual image, we would have to execute another asynchronous call on another background thread to download the image, and then update the ImageView on the main/ui thread. This is much more painful to do by hand than getting a simple JSON string is. We will use a library instead.

1. Look up an image-loading library for Android
   a. Glide and Picasso are common ones
2. Add it to the app/build.gradle file, in the *dependencies* segment

   ```
   // examples of Glide and Picasso. you only need one, and doesn't have to be these
   implementation "com.github.bumptech.glide:glide:4.8.0"
   implementation "com.squareup.picasso:picasso:2.71828"
   ```

3. Add code to the Button's click handler to start loading the image, using the smallRecipeUrl from the RecipeModel.
4. Commit and push your code. Use "bonus – image loading" as your commit message