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 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/AndroidLab3.git**
7. Change directory to the AndroidLab3 folder you just cloned
8. **Create a new branch** in the git repository called **LastnameFirstname**
9. Open Android Studio, and open the project

Part 1: AsyncTask

1. Open up *activity_main.xml* and look at the layout's Design, then examine the Text.
2. Note the elements already present in the layout:
    a. ConstraintLayout: the root layout for the associated Activity
    b. TextView (x2): an element for displaying text
        i. note this attribute on the second one: `tools:text="number of matches"`
    c. Button: an element allowing the user to invoke an action
    d. EditText: an element allowing users to provide text input

Anything with a `tools:` prefix affects the layout 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. If you launched the app in an emulator right now, you wouldn't see anything there.

3. Look in the <u>network</u> directory, and find the *RecipeSearchAsyncTask* class. This class lets us use a background thread to do long tasks, instead of potentially blocking the main thread.
   a. The *doInBackground* method is already filled out. It will run a search for recipes containing a certain ingredient, and return the number of matches that were found.

4. Notice the nested interface *RecipeListener*, which will let us handle the response back on the main thread over in *MainActivity*. This interface defines the contract between *RecipeSearchAsyncTask* and any class that signs up to listen for updates.

5. *RecipeSearchAsyncTask* needs to be able to keep an <u>implementation</u> of RecipeListener saved, so that it has something to notify when the task is complete.
   Add a private *RecipeListener* variable to the *RecipeSearchAsyncTask*, and add a setter.

6. Look in *MainActivity*. There should already be member variables declared that match the view elements from the *activity_main* layout. Add code to assign instances of the layout elements to the corresponding private class variables (using *findViewById*).

7. Add a listener to the button, so we can respond when the user clicks it:
```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

    }
});
```

OnClickListener is the same Listener pattern that we are building in *RecipeSearchAsyncTask*. The details of the contract are a little different – because it is doing a different job.
The main idea is the same though: the contract is an agreement between the Button and the MainActivity. The Button guarantees that when the user taps on the screen, it will call this *onClick* method. The MainActivity has to follow this exact format (method name, parameter, etc.) that the Button has declared.

8. In the Button click handler's *onClick* method, add this code:

```
// create a new task
RecipeSearchAsyncTask task = new RecipeSearchAsyncTask();

// create a Listener! (and add it to the task)
task.setRecipeListener(new RecipeSearchAsyncTask.RecipeListener() {
    @Override
    public void onRecipeCallback(String response) {
        // show response on the screen (in a TextView)
        responseText.setText(response);
    }
});

// start the task
String searchTerm = searchEditText.getText().toString();
task.execute(searchTerm);
```

This creates a concrete implementation of the *RecipeListener* 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 *responseText* TextView, to display the response on the screen.

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

Finally, the listener needs to be notified that the task is complete and a result is ready.

9. In the *onPostExecute* method, after the call to *super*, add one line of code that calls the *onRecipeCallback* function for the *RecipeSearchAsyncTask*'s private listener.

```
recipeListener.onRecipeCallback(result);
```

This will notify the concrete implementation of a RecipeListener that we created in *MainActivity* and added to the *RecipeSearchAsyncTask*.

Add breakpoints and/or logging output statements to the lab, and step through it until you are feeling confident that you understand the Listener pattern.

If you are stuck or unsure, ask for help!