CS 370 Lab 3: AsyncTask and Listeners

This lab is about running tasks on a background thread and using the Listener pattern.

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/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** (two): 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 network directory, and find the *RecipeCountAsyncTask* 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 an ingredient, and return the number of matches that were found.
    b. The *onPostExecute* method is called automatically after *doInBackground* completes.

4. Notice the nested interface *RecipeListener*, which will let us handle the response on the main thread over in *MainActivity*. The interface defines a contract between *RecipeCountAsyncTask* and any class that needs to listen to it for updates or results.

5. *RecipeCountAsyncTask* needs to be able to keep an instance of *RecipeListener* saved, so that it has something to notify when the task is complete.
   Add a private *RecipeListener* variable to the *RecipeCountAsyncTask*, and a setter method.

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 follows the same Listener pattern as the one we are building in *RecipeCountAsyncTask*. The details of the contract are a little different, because it is doing a different job.
   The main idea is the same: 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 provide an implementation of this method following the exact format (methodname/parameters/returntype) that the Button has defined. The definition of this method is in the *OnClickListener* interface, nested in the *View* class.

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

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

// create a Listener! (and add it to the task)
task.setListener(new RecipeCountAsyncTask.RecipeListener() {
    @Override
    public void onRecipeCallback(Integer response) {
        // show response on the screen (in a TextView)
        responseText.setText("Number of matches: " + response);
    }
});

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

   This creates an instance of the RecipeListener interface. If you peek back at where the interface is defined, you should see that it only contains one method (an abstract method, because interface!). The concrete implementation of that method here contains code that sets the text attribute of the *responseText* TextView.

   When the button is clicked, we make a new *RecipeCountAsyncTask* for background thread operations, attach our *RecipeListener* to it, and then execute the *doInBackground* function.

When the task is complete, the listener needs to be notified and passed the result.

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

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

   This will call into the concrete implementation of a RecipeListener that we created in *MainActivity*, and pass it the `result` from our *AsyncTask* work.
   The concrete implementation in *MainActivity* can then use the result to update the UI.

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

If you are stuck or unsure, ask for help!