

Lesson 2: Connect Sunshine to the Cloud

Free weather API for developers: openweathermap.org/API

The weather in London:

<http://openweathermap.org/current>

then in the catalog on the right, choose Other Features, Search Accuracy,

Accurate api.openweathermap.org/data/2.5/find?q=London&type=accurate&mode=xml

or directly go to

[http://api.openweathermap.org/data/2.5/find?](http://api.openweathermap.org/data/2.5/find?q=London&type=accurate&mode=xml&appid=2de143494c0b295cca9337e1e96b00e0)

[q=London&type=accurate&mode=xml&appid=2de143494c0b295cca9337e1e96b00e0](http://api.openweathermap.org/data/2.5/find?q=London&type=accurate&mode=xml&appid=2de143494c0b295cca9337e1e96b00e0)

Choose the metric:

<http://openweathermap.org/current>

In the catalog on the right, choose Other Features, Units format, metric. Then can change the name of the city in the URL.

URL query for Sunshine app: 1 week, postal code 94043, JSON, metric. The answer provided by the video does not work:

<http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode=json&units=metric&cnt=7>

Tao figured out that we should append something more. The following works:

[http://api.openweathermap.org/data/2.5/forecast/daily?](http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode=json&units=metric&cnt=7&appid=2de143494c0b295cca9337e1e96b00e0)

[q=94043&mode=json&units=metric&cnt=7&appid=2de143494c0b295cca9337e1e96b00e0](http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode=json&units=metric&cnt=7&appid=2de143494c0b295cca9337e1e96b00e0)

HTTP request for weather:

1. Make HTTP request
2. Read response from input stream
3. Clean up and log any errors

Code provided:

<https://gist.github.com/udacityandroid/d6a7bb21904046a91695>

(copied below)

```
1      // These two need to be declared outside the try/catch
2      // so that they can be closed in the finally block.
3      HttpURLConnection urlConnection = null;
4      BufferedReader reader = null;
5
6      // Will contain the raw JSON response as a string.
7      String forecastJsonStr = null;
8
9      try {
10         // Construct the URL for the OpenWeatherMap query
11         // Possible parameters are available at OWM's forecast API page, at
12         // http://openweathermap.org/API#forecast
13         URL url = new URL("http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode
14
15         // Create the request to OpenWeatherMap, and open the connection
16         urlConnection = (HttpURLConnection) url.openConnection();
17         urlConnection.setRequestMethod("GET");
18         urlConnection.connect();
19
20         // Read the input stream into a String
21         InputStream inputStream = urlConnection.getInputStream();
22         StringBuffer buffer = new StringBuffer();
23         if (inputStream == null) {
24             // Nothing to do.
25             return null;
26         }
27         reader = new BufferedReader(new InputStreamReader(inputStream));
28     }
```

```

28
29     String line;
30     while ((line = reader.readLine()) != null) {
31         // Since it's JSON, adding a newline isn't necessary (it won't affect parsing)
32         // But it does make debugging a *lot* easier if you print out the completed
33         // buffer for debugging.
34         buffer.append(line + "\n");
35     }
36
37     if (buffer.length() == 0) {
38         // Stream was empty. No point in parsing.
39         return null;
40     }
41     forecastJsonStr = buffer.toString();
42 } catch (IOException e) {
43     Log.e("PlaceholderFragment", "Error ", e);
44     // If the code didn't successfully get the weather data, there's no point in attempting
45     // to parse it.
46     return null;
47 } finally{
48     if (urlConnection != null) {
49         urlConnection.disconnect();
50     }
51     if (reader != null) {
52         try {
53             reader.close();
54         } catch (final IOException e) {
55             Log.e("PlaceholderFragment", "Error closing stream", e);
56         }
57     }
58 }
59
60     return rootView;
61 }
62 }
63 }

```

In the sample code provided you'll notice that we create a Http Url connection (Tao: HttpURLConnection class). To use Http to send and receive data over the network, we have two clients on Android. The Http Url connection class, as well as the patchy(縫補的) Http Client class. Both options support Https, streaming uploads and downloads, configurable time outs, IPB 6 and connection pooling. We recommend the Http Url Connection class because it's general purpose and light weight, and it's been optimized to suit the needs of most Android apps.

From <http://openweathermap.org/appid#use>

To get access to weather API you need an API key whatever account you chose from Free to Enterprise.

API call:

<http://api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID={APIKEY}>

Parameters:

APPID {APIKEY} is your unique API key

Example of API call:

`api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID=1111111111`

My API key: `f0c9dc0bef32981ff3f3e77d4b7af077`

Drop the above code into the sunshine app to query for actual weather data. Open up your project in `main activity.java`. In this class, scroll down to the placeholder fragment class (i.e., `MainActivityFragment.java` out of `MainActivity` in my Android Studio), within the placeholder fragment class in the `on create view` method, scroll down and go ahead and add the networking code snippet(片断) here (Tao: added after the line of `listView.setAdapter(forecastAdapter)`). You may need to enable these options in auto import or add the imports manually. Tao: the way is the following: File, Settings, Editor, General, Auto Import, Java. Check all the boxes and insert all imports on paste (should mean to select All in Insert imports on paste).

When you run the app, you'll see that it crashes. After using logcat to figure out the error (omitted here by Tao), we found that it is `NetWorkOnMainException`. It's within the placeholder fragment class, in the `onCreateView` method. And it happens in `main activity.java` file, in line 116. So if we go back to the code, on that line you can see that `urlConnection.connect()` actually caused the error, and that we can't do that on the main thread.

When we said that the framework didn't want us to run network operations on the main thread, what is the main thread? Well, Android apps run by default on the main thread, also called the UI thread. It handles all the user input as well as the output, such as screen drawing. Thus we want to avoid any time-consuming operations here, otherwise the UI (Tao: sounds like UI) is going to stutter(結結巴巴). Instead, kick off a background worker thread if you have to do some long-running work. This includes doing network calls, decoding bitmaps, or reading, and writing from the database. Okay. So, somehow, we have to move the networking code off the main thread. But how are we going to do that? Well there are several options ...

What class does Android use to simplify background thread creation and UI thread synchronization? `AsyncTask`. For example, when someone clicks a button, then you just initialize the task, and then you can call `execute` on it, and then pass in any parameters that are needed. Notice that when you're extending the `Async` class, there's a couple of generics that you need to specify. The first is the type that will be passed into the `do in background` method. So, if you want to pass in this image URL that is specified string here and then in `doing background` you'll get a string parameter. Then second one is for the type of object that you'll get when you get progress updates as a task gets executed. We're not using that here, so it's okay to specify that as `void`. And the third type is type of results that we'll be sending back to the main thread through the `onPostExecute` method.

For example, you can implement the previous example using `AsyncTask` this way:

```
public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a worker thread and
     * delivers it the parameters given to AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    /** The system calls this to perform work in the UI thread and delivers
     * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

`onPreExecute` happens on the main thread. And here you can do any setup work. Then `doInBackground` happens on the background thread. While this is running, you can actually call `publishProgress` as many times as you want, so that you can pass information to the UI. So that it can update and then tell the user that a certain percentage of the work is done. Each time this is called, it triggers `onProgressUpdate` with some information. Then, you can show a loading indicator in your UI that says something's 10% done, 50% done, 100% done. And all this happens on the main thread. And then, once all of this is complete in the background thread, then it calls on `PostExecute` with the results on the main thread.

AsyncTask

MAIN or BACKGROUND thread?

M B

●	○	onPreExecute()	
○	●	doInBackground()	— can call publishProgress() here
●	○	onProgressUpdate()	
●	○	onPostExecute()	

Let's apply what we just learned by opening up the MainActivity.java file within our project. We're going to take this networking code snippet and move it over to its own AsyncTask, so it runs in a background thread. The task is going to be defined within this fragment class. But speaking of which, it's actually still called PlaceholderFragment. Let's do a little bit of refactoring now by giving it a real name.

Actually a more robust way to change the name of the PlaceholderFragment is to use the "Refactor" tool provided by Android Studio. Right click on the class name PlaceholderFragment > Refactor > Rename > type in ForecastFragment and then all relevant references will be updated.

The name PlaceholderFragment → ForecastFragment, and then update it in other appropriate places as well. Move ForecastFragment in to its own file, that way the MainActivity won't get so long and cumbersome. Within ForecastFragment, define a new inner class called FetchWeatherTask which extends from AsyncTask:

In ForecastFragment.java:

.....

```

.....
return rootView;
}

public class FetchWeatherTask {

}

}

```

And then you can move the networking code snippet here. Now the app does not crash now.

Move to AsyncTask

Time to code!



- ☐ Rename PlaceholderFragment to ForecastFragment
- ☐ Move ForecastFragment to new file
- ☐ Create AsyncTask called FetchWeatherTask with networking code snippet

The code can be found in

https://github.com/udacity/Sunshine-Version-2/compare/2.01_add_network_code...2.02_refactor_forecast_fragment

What Tao did as extra:

1. Changed the first line in ForecastFragment.java from
 package com.example.android.sunshine.app;
 to
 package com.example.android.sunshine;

2. Changed the line
 String apiKey = "&APPID=" + BuildConfig.OPEN_WEATHER_MAP_API_KEY;
 to
 String apiKey = "&APPID=f0c9dc0bef32981ff3f3e77d4b7af077";

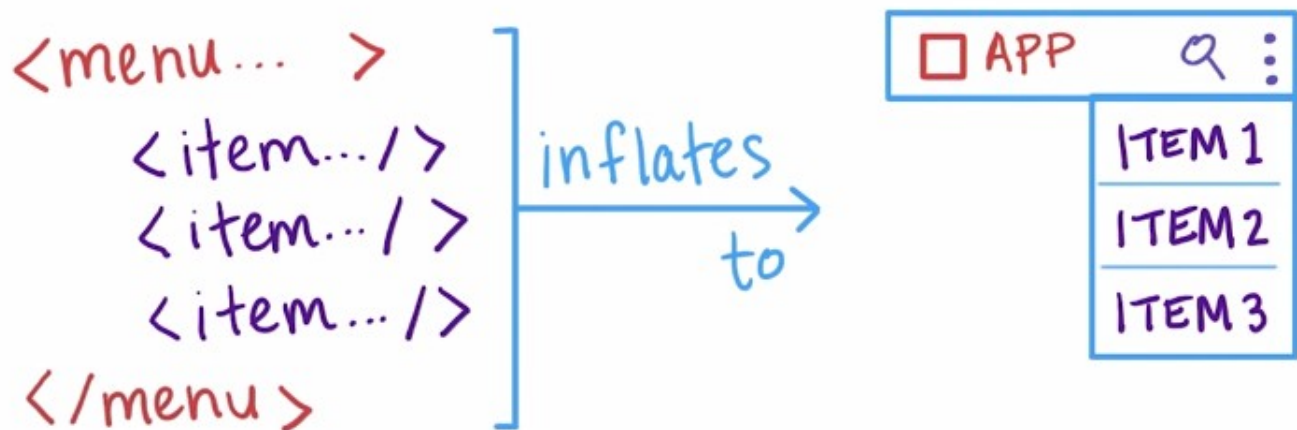
3. Did not add `buildTypes.each{...}` in `app/build.gradle`, as shown in the above web address. If add this, a compiling error occurred.

I also used the command line to compile, because my phone could not open the developer mode.

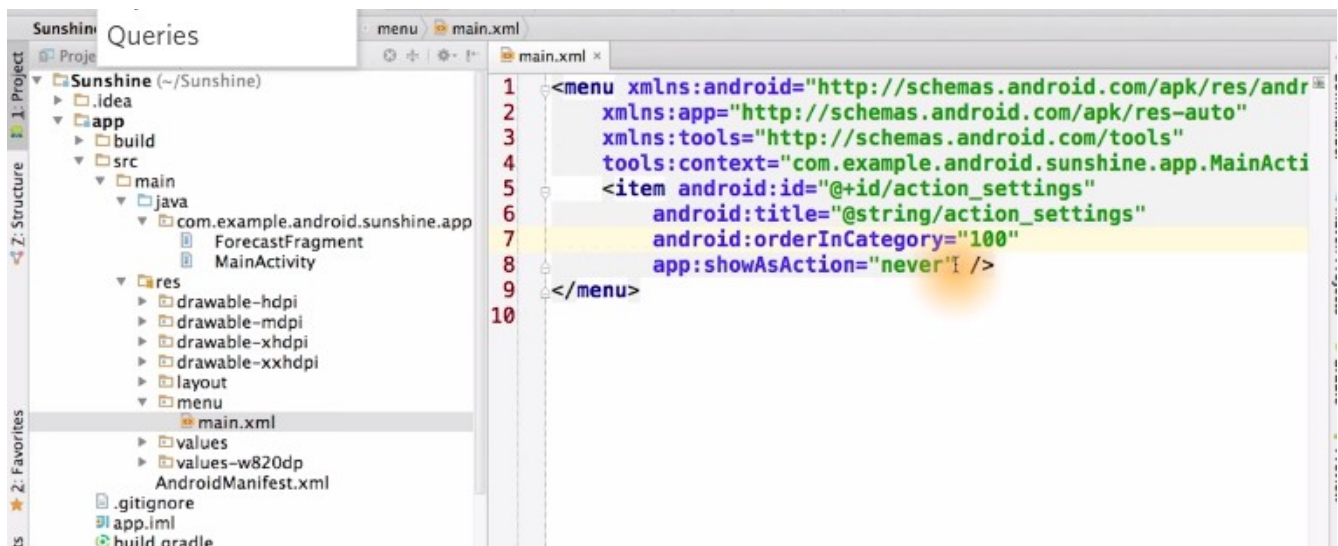
For debugging purposes, it will be nice if we can execute the task any time we wanted by interacting with the UI somehow. So, we're going to add a refresh menu option for debugging. A warning, though, this menu option should not shift in the final app.

Menu options:

With that in mind let's add a Refresh Menu button to our app. On Android, menu options are defined in XML and they can be declared for fragments or activities. When the fragment or activity is created, it inflates (結合下圖, 可深刻理解 inflate 之意思, 即 XML 是很簡單的, inflate 就是產生完整的 Java 代碼(及 app 中的真實 menu)) this XML into the actual menu items in the app. You'll see that there are Action buttons (如下圖中的放大鏡) which are menu items that appear in the Action bar (即放大鏡所在的那個矩形框), such as this Search Menu item. This space (即放大鏡所在的那個矩形框) is reserved for the most prominent actions in your app. Then anything else that's less important falls into the overflow menu by tapping on this button with the three dots. These menu items are ordered from most frequently used to least frequently used. And on larger devices that have more screen real estate, you can specify that some of these menu items can actually go into the Action bar if there's room.

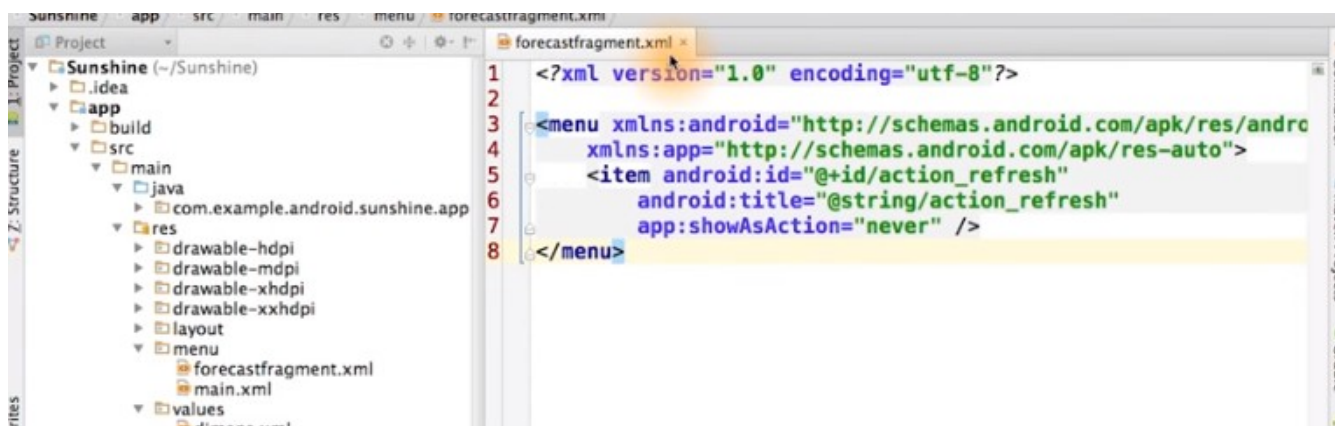


If you go back to our project in the Resources folder which is called `res`, there is a `Menu` folder and inside that there's a `main.xml` file. If you open that up you see the menu layout XML, and that there's a single menu option defined for Settings. It will never show up as an action in the Action bar, meaning that it will be in the Overflow menu. You can verify this by checking the app on your phone. To define the ordering of menu items, you can just add multiple items to this XML file, and then they will show up in that order in the app. If you don't like the order though, and you want to explicitly set it, then you can specify this order in `Category`, `value`. Right now it's set as 100, so that the Settings menu will be at the bottom of all the other menu options that we define in our app. The only Menu option that should show up below the Settings menu is the Help menu.

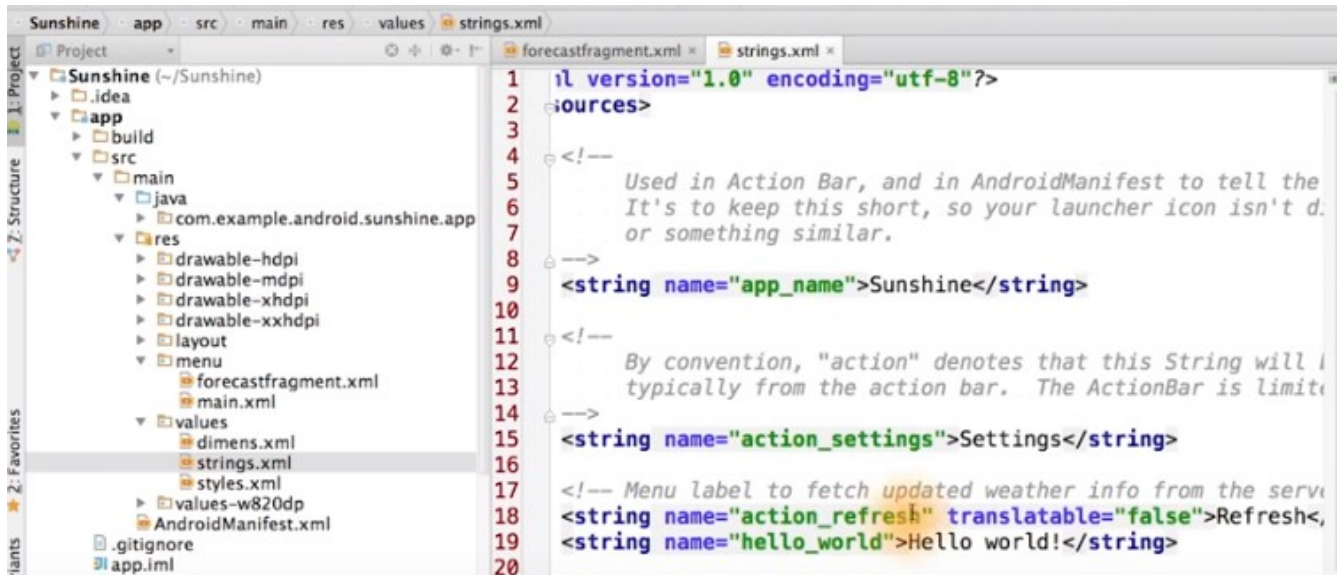


REFRESH BUTTON

Create new menu layout forecastfragment.xml
with menu item that has ID **action_refresh**
with label **Refresh**



(the above graph) We want it to appear in the overflow menu, so we set showAsAction as never. We also declare the string. You can find the string here in the strings.xml file, which is in res/values. Note that forecastfragment.xml should be in the folder res/menu, not in res/layout!



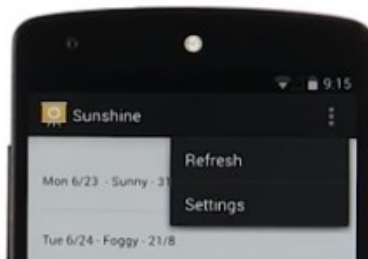
The screenshot shows the Android Studio IDE. On the left, the Project Structure pane displays the hierarchy of the 'Sunshine' app, including the 'res' directory and the 'strings.xml' file. The main editor window shows the content of 'strings.xml' with the following code:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3
4 <!--
5     Used in Action Bar, and in AndroidManifest to tell the
6     system the icon to use. It's to keep this short, so your launcher icon isn't d.
7     or something similar.
8 -->
9 <string name="app_name">Sunshine</string>
10
11 <!--
12     By convention, "action" denotes that this String will be used typically from the action bar. The ActionBar is limited
13     in the number of items it can hold, so you can use this to keep the number of items low.
14 -->
15 <string name="action_settings">Settings</string>
16
17 <!-- Menu label to fetch updated weather info from the server -->
18 <string name="action_refresh" translatable="false">Refresh</string>
19 <string name="hello_world">Hello world!</string>
20
```

You can get your strings.xml file translated into different languages. You can mark strings that you don't need translated using `translatable=false`.

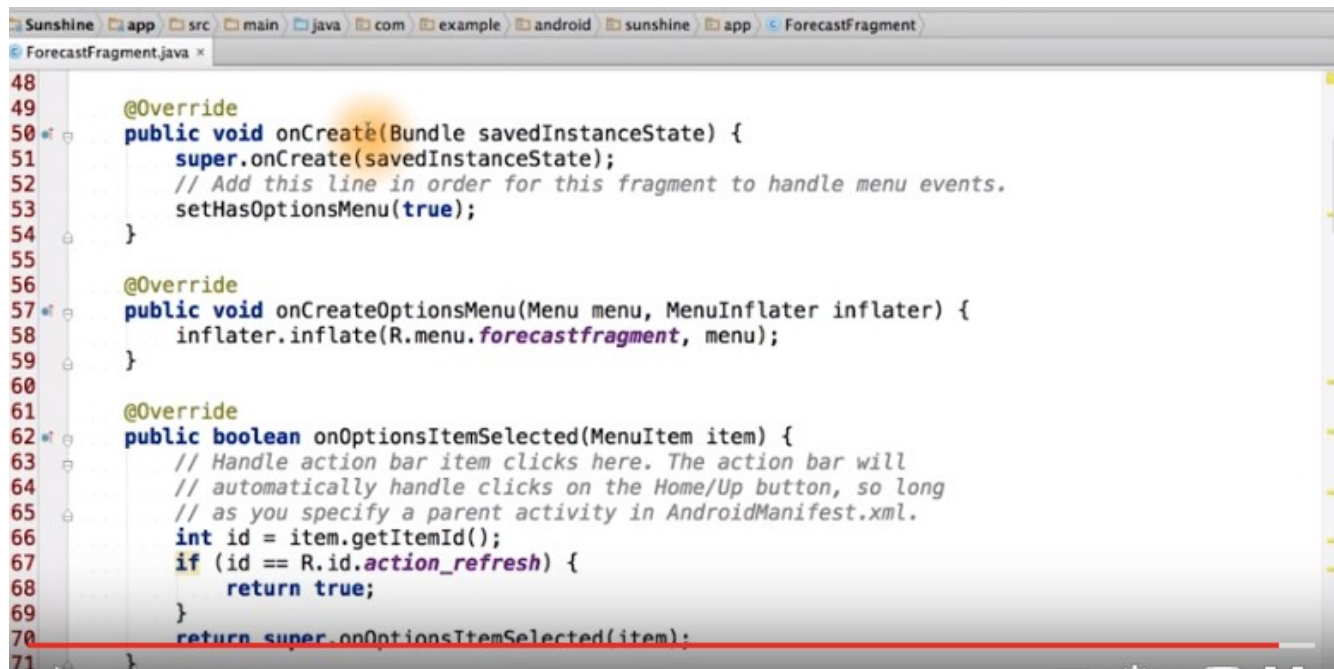
Refresh Button

1. Inflate options menu in `ForecastFragment`
2. Set up `onOptionsItemSelected()`



What method in **Fragment** do you call to report that it has menu options?

`Fragment.setHasOptionsMenu(true)`



```
48
49
50 @Override
51 public void onCreate(Bundle savedInstanceState) {
52     super.onCreate(savedInstanceState);
53     // Add this line in order for this fragment to handle menu events.
54     setHasOptionsMenu(true);
55 }
56
57 @Override
58 public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
59     inflater.inflate(R.menu.forecastfragment, menu);
60 }
61
62 @Override
63 public boolean onOptionsItemSelected(MenuItem item) {
64     // Handle action bar item clicks here. The action bar will
65     // automatically handle clicks on the Home/Up button, so long
66     // as you specify a parent activity in AndroidManifest.xml.
67     int id = item.getItemId();
68     if (id == R.id.action_refresh) {
69         return true;
70     }
71     return super.onOptionsItemSelected(item);
72 }
```

Now modify the code so that when the Refresh menu item is selected, it executes the FetchWeathTask:

In ForecastFragment.java:

```
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
    if (id == R.id.action_refresh) {
        FetchWeatherTask weatherTask = new FetchWeatherTask(); //此句是新加的
        weatherTask.execute(); //此句是新加的
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

When you tap on the Refresh button, it actually crashes. What error is causing this? It's SecurityException. The log says Permission denied and ask if you're missing the INTERNET permission or not. And indeed, we are missing the INTERNET permission, so we need to request it.

Which of the following activities can only be done after declaring a user-permission in the manifest?

Taking a photo

Making a phone call

(correct) Getting the user's current location

Accessing a user-selected contact's details

What permission must you add to your code to declare the Internet permission in your AndroidManifest.xml? The answer is android.permission.INTERNET. Open main/AndroidManifest.xml. In this file, we declare uses permission, with the name of the permission.

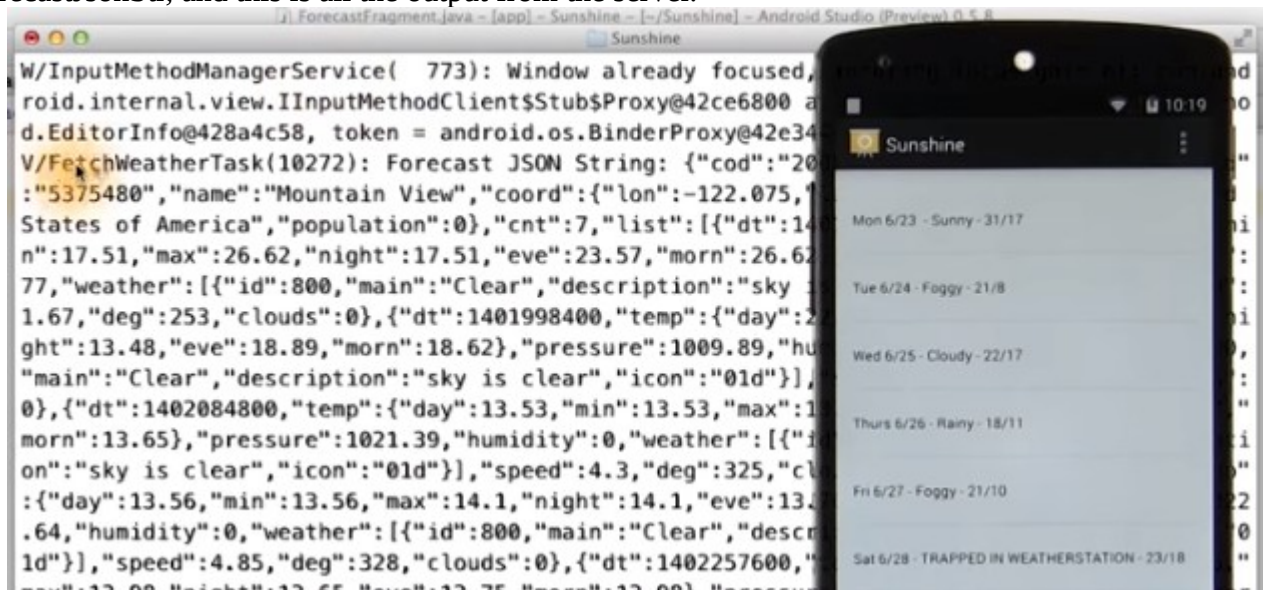
```
Sunshine - app - src - main - AndroidManifest.xml
AndroidManifest.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.example.android.sunshine.app" >
4
5     <!-- This permission is necessary in order for Sunshine to perform network access. -->
6     <uses-permission android:name="android.permission.INTERNET" />
7
8     <application
9         android:allowBackup="true"
```

We verify that the data returned is correct by adding a verbose log statement for printing out the forecastJsonStr (應該就是天氣網頁上的那個 string). This is in the fetch weather task.

```
Sunshine - app - src - main - java - com - example - android - sunshine - app - ForecastFragment
ForecastFragment.java
126     }
127     reader = new BufferedReader(new InputStreamReader(inputStream));
128
129     String line;
130     while ((line = reader.readLine()) != null) {
131         // Since it's JSON, adding a newline isn't necessary (it won't affect p
132         // But it does make debugging a *lot* easier if you print out the compl
133         // buffer for debugging.
134         buffer.append(line + "\n");
135     }
136
137     if (buffer.length() == 0) {
138         // Stream was empty. No point in parsing.
139         return null;
140     }
141     forecastJsonStr = buffer.toString();
142
143     Log.v(LOG_TAG, "Forecast JSON String: " + forecastJsonStr);
144
145     } catch (IOException e) {
```

We can verify that it's going to show up in the logs. For this example, I'm going to use the command line (type: adb logcat). There are the real time logs, and then if I hit refresh (Tao: on the phone), then I

see the weather data here. The verbose log, and this is our log tag fetch weather task. Here's the forecastJsonStr, and this is all the output from the server.



Great work. Thumbs up. The data has landed safely on our phones.

We'd really like our user to be able to change their location in the settings. So let's make the FetchWeatherTask more flexible by having it take as input a postal code parameter. We should use a UriBuilder Class to build up the URL. We can declare a base URL and then append each pair of query PARAM and PARAM values onto it. This includes PARAM's for post code, JSON format, metric units, and date count.

POSTAL CODE PARAM

- ☐ Change **FetchWeatherTask** to take postcode as input param
- ☐ Use **UriBuilder** to build up the URL

We want to modify the FetchWeatherTask, so that we can pass in the postal code parameter when we execute that task. Here you see that we modified the generic input type, to be a string so that the doInBackground method receives string params. Now, we only passed in one string param, 94043. So when you see down here later, we can read out the zeroth position in the params array, to get that postal code. Notice that we're using the UI builder here, and we're appending query parameters, one by one.

We define constants for these query params, as seen here. And for the values, we also define them up there.

下圖來自 Github, 好方便看改動的地方.

注意在 figure1/4 中, 我的實際添加的代碼為:

```
buildTypes.each {  
    it.buildConfigField 'String', 'OPEN_WEATHER_MAP_API_KEY',  
    "\"f0c9dc0bef32981ff3f3e77d4b7af077\""  
}
```

否不按以上方式寫, 則編譯時會報錯, 以上解決方法來自(不用看):

<http://stackoverflow.com/questions/33365650/cannot-resolve-symbol-c882c94be45fff9d16a1cf845fc16ec5>

figure 1/4:

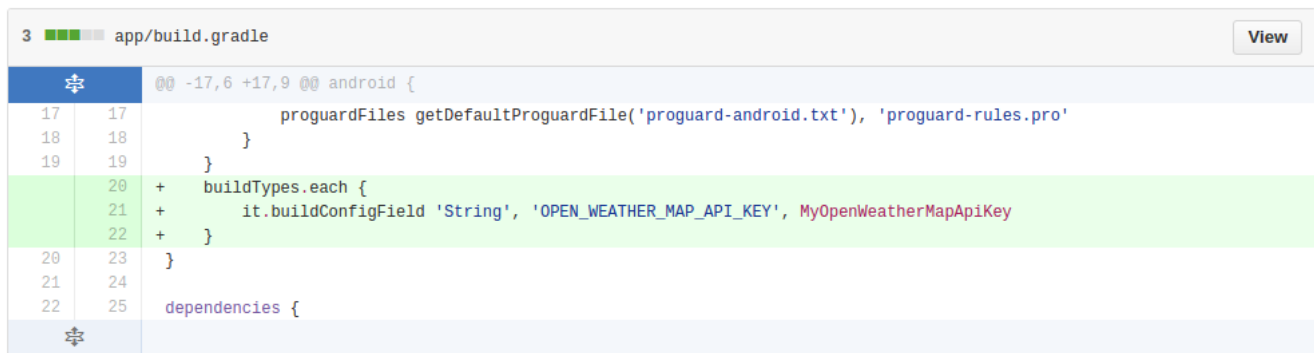


figure 2/4:

p/src/main/java/com/example/android/sunshine/app/ForecastFragment.java

```
@@ -15,6 +15,7 @@
```

```
*/
```

```
package com.example.android.sunshine.app;
```

```
+import android.net.Uri;
```

```
import android.os.AsyncTask;
```

```
import android.os.Bundle;
```

```
import android.support.v4.app.Fragment;
```

```
@@ -68,7 +69,7 @@ public boolean onOptionsItemSelected(MenuItem item) {
```

```
    int id = item.getItemId();
```

```
    if (id == R.id.action_refresh) {
```

```
        FetchWeatherTask weatherTask = new FetchWeatherTask();
```

```
-        weatherTask.execute();
```

```
+        weatherTask.execute("94043");
```

```
        return true;
```

```
    }
```

```
    return super.onOptionsItemSelected(item);
```

```
@@ -109,12 +110,18 @@ public View onCreateView(LayoutInflater inflater, ViewGroup container,
```

```
    return rootView;
```

```
}
```

```
- public class FetchWeatherTask extends AsyncTask<Void, Void, Void> {
```

```
+ public class FetchWeatherTask extends AsyncTask<String, Void, Void> {
```

```
    private final String LOG_TAG = FetchWeatherTask.class.getSimpleName();
```

figure 3/4:


```

@Override
protected Void doInBackground(Void... params) {
protected Void doInBackground(String... params) {

    // If there's no zip code, there's nothing to look up. Verify size of params.
    if (params.length == 0) {
        return null;
    }

    // These two need to be declared outside the try/catch
    // so that they can be closed in the finally block.
    HttpURLConnection urlConnection = null;

@@ -123,11 +130,33 @@ protected Void doInBackground(Void... params) {
    // Will contain the raw JSON response as a string.
    String forecastJsonStr = null;

    String format = "json";
    String units = "metric";
    int numDays = 7;

    try {
        // Construct the URL for the OpenWeatherMap query
        // Possible parameters are available at OWM's forecast API page, at
        // http://openweathermap.org/API#forecast
-        URL url = new URL("http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode=json&units=metric
+        final String FORECAST_BASE_URL =
+            "http://api.openweathermap.org/data/2.5/forecast/daily?";
+        final String QUERY_PARAM = "q";
+        final String FORMAT_PARAM = "mode";
+        final String UNITS_PARAM = "units";
+        final String DAYS_PARAM = "cnt";
+        final String APPID_PARAM = "APPID";

```

figure 4/4:

```

+
+        Uri builtUri = Uri.parse(FORECAST_BASE_URL).buildUpon()
+            .appendQueryParameter(QUERY_PARAM, params[0])
+            .appendQueryParameter(FORMAT_PARAM, format)
+            .appendQueryParameter(UNITS_PARAM, units)
+            .appendQueryParameter(DAYS_PARAM, Integer.toString(numDays))
+            .appendQueryParameter(APPID_PARAM, BuildConfig.OPEN_WEATHER_MAP_API_KEY)
+            .build();
+
+        URL url = new URL(builtUri.toString());
+
+        Log.v(LOG_TAG, "Built URI " + builtUri.toString());

    // Create the request to OpenWeatherMap, and open the connection
    urlConnection = (HttpURLConnection) url.openConnection();

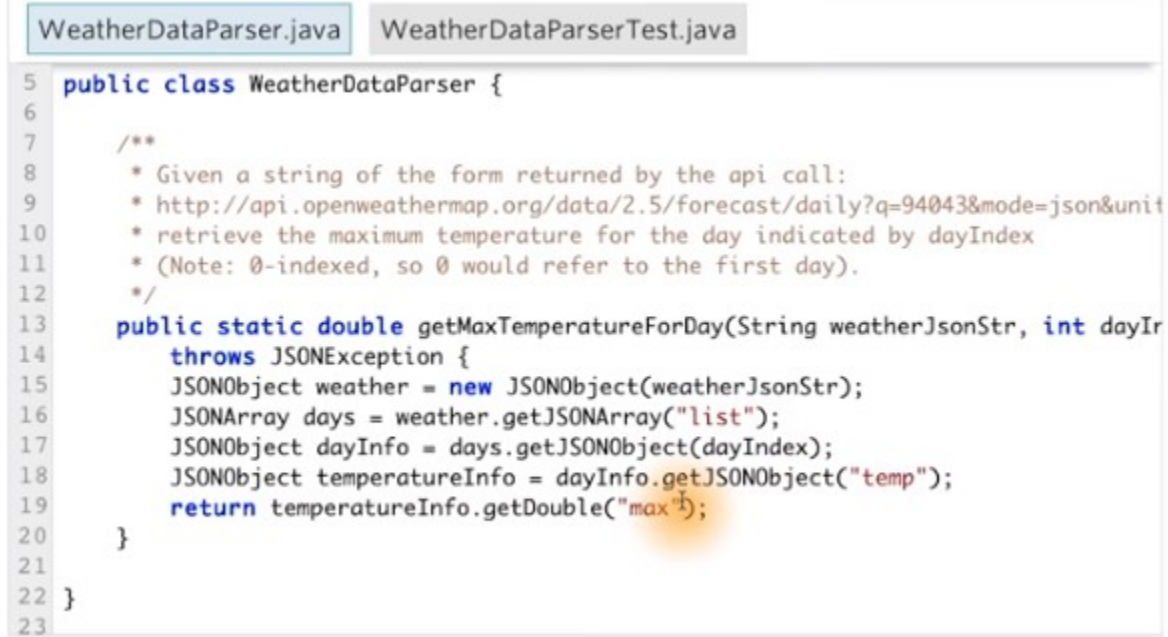
```

The URL that we built looks good, and the JSON string from the server also looks good, however, it still is one long string. Let's look at it more carefully to see what data, we should extract from it.

Desired JSON attributes. Which attributes do we care about for Sunshine? Answer: min, max, main. You might be wondering why time is not included. The weather API provides the dat and GMT time, but doesn't provide time zones for the location. So unfortunately, the date could be incorrect in some cases depending on your location and time of day. Instead, for a more accurate experience, we'll be computing the date ourselves based on the device time of when you fetched the weather forecast from

the server.

For the next step, let's do an exercise that focuses on extracting out the value of one of those attributes.



```
5 public class WeatherDataParser {
6
7     /**
8      * Given a string of the form returned by the api call:
9      * http://api.openweathermap.org/data/2.5/forecast/daily?q=94043&mode=json&unit
10     * retrieve the maximum temperature for the day indicated by dayIndex
11     * (Note: 0-indexed, so 0 would refer to the first day).
12     */
13     public static double getMaxTemperatureForDay(String weatherJsonStr, int dayIndex)
14         throws JSONException {
15         JSONObject weather = new JSONObject(weatherJsonStr);
16         JSONArray days = weather.getJSONArray("list");
17         JSONObject dayInfo = days.getJSONObject(dayIndex);
18         JSONObject temperatureInfo = dayInfo.getJSONObject("temp");
19         return temperatureInfo.getDouble("max");
20     }
21 }
22
23
```

Full code:

fig 1/6:

app/src/main/java/com/example/android/sunshine/app/ForecastFragment.java

@@ -19,6 +19,7 @@

```
import android.os.AsyncTask;
import android.os.Bundle;
import android.support.v4.app.Fragment;
+import android.text.format.Time;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.Menu;
```

@@ -29,12 +30,17 @@

```
import android.widget.ArrayAdapter;
import android.widget.ListView;
```

```
+import org.json.JSONArray;
+import org.json.JSONException;
+import org.json.JSONObject;
+
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
```

```
+import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
@@ -110,12 +116,110 @@ public View onCreateView(LayoutInflater inflater, ViewGroup container,
    return rootView;
}
```

fig 2/6:

```

- public class FetchWeatherTask extends AsyncTask<String, Void, Void> {
+ public class FetchWeatherTask extends AsyncTask<String, Void, String[]> {

    private final String LOG_TAG = FetchWeatherTask.class.getSimpleName();

+    /** The date/time conversion code is going to be moved outside the async task later,
+     * so for convenience we're breaking it out into its own method now.
+     */
+    private String getReadableDateString(long time){
+        // Because the API returns a unix timestamp (measured in seconds),
+        // it must be converted to milliseconds in order to be converted to valid date.
+        SimpleDateFormat shortenedDateFormat = new SimpleDateFormat("EEE MMM dd");
+        return shortenedDateFormat.format(time);
+    }
+
+    /**
+     * Prepare the weather high/lows for presentation.
+     */
+    private String formatHighLows(double high, double low) {
+        // For presentation, assume the user doesn't care about tenths of a degree.
+        long roundedHigh = Math.round(high);
+        long roundedLow = Math.round(low);
+
+        String highLowStr = roundedHigh + "/" + roundedLow;
+        return highLowStr;
+    }

```

fig 3/6:

```

+  /**
+   * Take the String representing the complete forecast in JSON Format and
+   * pull out the data we need to construct the Strings needed for the wireframes.
+   *
+   * Fortunately parsing is easy:  constructor takes the JSON string and converts it
+   * into an Object hierarchy for us.
+   */
+  private String[] getWeatherDataFromJson(String forecastJsonStr, int numDays)
+      throws JSONException {
+
+      // These are the names of the JSON objects that need to be extracted.
+      final String OWM_LIST = "list";
+      final String OWM_WEATHER = "weather";
+      final String OWM_TEMPERATURE = "temp";
+      final String OWM_MAX = "max";
+      final String OWM_MIN = "min";
+      final String OWM_DESCRIPTION = "main";
+
+      JSONObject forecastJson = new JSONObject(forecastJsonStr);
+      JSONArray weatherArray = forecastJson.getJSONArray(OWM_LIST);
+
+      // OWM returns daily forecasts based upon the local time of the city that is being
+      // asked for, which means that we need to know the GMT offset to translate this data
+      // properly.
+
+      // Since this data is also sent in-order and the first day is always the
+      // current day, we're going to take advantage of that to get a nice
+      // normalized UTC date for all of our weather.
+
+      Time dayTime = new Time();
+      dayTime.setToNow();
+
+      // we start at the day returned by local time. Otherwise this is a mess.
+      int julianStartDay = Time.getJulianDay(System.currentTimeMillis(), dayTime.gmtoff);
+
+      // now we work exclusively in UTC
+      dayTime = new Time();

```


fig 4/6:

```

+ String[] resultStrs = new String[numDays];
+ for(int i = 0; i < weatherArray.length(); i++) {
+     // For now, using the format "Day, description, hi/low"
+     String day;
+     String description;
+     String highAndLow;
+
+     // Get the JSON object representing the day
+     JSONObject dayForecast = weatherArray.getJSONObject(i);
+
+     // The date/time is returned as a long. We need to convert that
+     // into something human-readable, since most people won't read "1400356800" as
+     // "this saturday".
+     long dateTime;
+     // Cheating to convert this to UTC time, which is what we want anyhow
+     dateTime = dayForecast.getLong("date");
+     day = getReadableDateString(dateTime);
+
+     // description is in a child array called "weather", which is 1 element long.
+     JSONArray weatherArray = dayForecast.getJSONArray("weather");
+     JSONObject weatherObject = weatherArray.getJSONObject(0);
+     description = weatherObject.getString("description");
+
+     // Temperatures are in a child object called "temp". Try not to name variables
+     // "temp" when working with temperature. It confuses everybody.
+     JSONObject temperatureObject = weatherObject.getJSONObject("temp");
+     double high = temperatureObject.getDouble("high");
+     double low = temperatureObject.getDouble("low");
+
+     highAndLow = formatHighLows(high, low);
+     resultStrs[i] = day + " - " + description + " - " + highAndLow;
+ }
+
+ for (String s : resultStrs) {
+     Log.v(LOG_TAG, "Forecast entry: " + s);
+ }
+
+ return resultStrs;
+
+ }

```

fig 5/6:

```

@Override
- protected Void doInBackground(String... params) {
+ protected String[] doInBackground(String... params) {

    // If there's no zip code, there's nothing to look up. Verify size of params.
    if (params.length == 0) {
@@ -144,12 +248,14 @@ protected Void doInBackground(String... params) {
        final String FORMAT_PARAM = "mode";
        final String UNITS_PARAM = "units";
        final String DAYS_PARAM = "cnt";
+       final String APPID_PARAM = "APPID";

        Uri builtUri = Uri.parse(FORECAST_BASE_URL).buildUpon()
            .appendQueryParameter(QUERY_PARAM, params[0])
            .appendQueryParameter(FORMAT_PARAM, format)
            .appendQueryParameter(UNITS_PARAM, units)
            .appendQueryParameter(DAYS_PARAM, Integer.toString(numDays))
+           .appendQueryParameter(APPID_PARAM, BuildConfig.OPEN_WEATHER_MAP_API_KEY)
            .build();

        URL url = new URL(builtUri.toString());
@@ -183,6 +289,8 @@ protected Void doInBackground(String... params) {
        return null;
    }
    forecastJsonStr = buffer.toString();
+
+    Log.v(LOG_TAG, "Forecast string: " + forecastJsonStr);
} catch (IOException e) {
    Log.e(LOG_TAG, "Error ", e);
    // If the code didn't successfully get the weather data, there's no point in attempting
@@ -200,6 +308,15 @@ protected Void doInBackground(String... params) {
    }
}
}
}

```

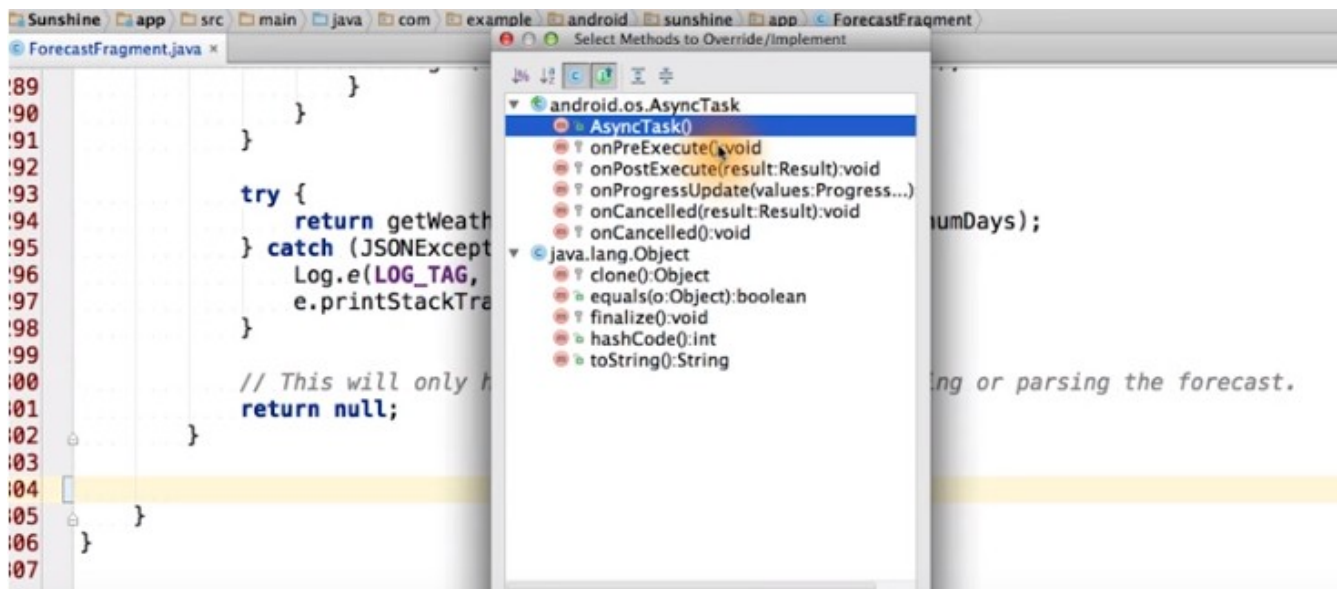
fig 6/6:

```

+     try {
+         return getWeatherDataFromJson(forecastJsonStr, numDays);
+     } catch (JSONException e) {
+         Log.e(LOG_TAG, e.getMessage(), e);
+         e.printStackTrace();
+     }
+
+     // This will only happen if there was an error getting or parsing the forecast.
+     return null;
+ }
}

```

From the logs, we know that we have the right forecast data and it's in the right format that we want as an array of strings. So it's finally time to update the UI. Think back on how `AsyncTasks` are able to pass data back onto the main thread. You can hit `Ctrl+O` to see the list of available methods we can override in `AsyncTask` (Tao: the mouse should be at (and click) the “`AsyncTask`” keyword in the line “`public class FetchWeatherTask extends AsyncTask`”). If you click on any of them, it will be prepopulated in the code for you.



Then, you can update `ArrayAdapter` with the new data that was retrieved by the `AsyncTask`. As a hint, you can make the `ForecastAdapter` be a global variable.

```
public class ForecastFragment extends Fragment {

    private ArrayAdapter<String> mForecastAdapter;

    public ForecastFragment() {
    }

    @Override
```

That way, you can access it from within the `FetchWeatherTask`. Make sure that this is not a static class, otherwise, you won't be able to access the member variable from the forecast fragment.

```
public class FetchWeatherTask extends AsyncTask<String, Void, String[]> {
```

Tao: all the above changes are already included in the code in the earlier figures 1/6-6/6.

The solution is to override the `onPostExecute` method of the `AsyncTask`. And this runs on the main thread. We received the string array of forecast results, which came as a return value from the `doInBackground` method above. First, we clear the `ForecastAdapter` of all the previous forecast entries. Then we go ahead and add each new forecast entry one by one to the `ForecastAdapter`. This is what ultimately triggers the `ListView` to the update. Once you hit Refresh, you'll see the weather forecast for the next seven days for your location. Even though we hit Refresh, we don't have any verbose logging statements being printed out here. We don't need them anymore....

```

267     }
268     // This will only happen if there was an error getting or parsing the forecast.
269     return null;
270 }
271
272
273 @Override
274 protected void onPostExecute(String[] result) {
275     if (result != null) {
276         mForecastAdapter.clear();
277         for (String dayForecastStr : result) {
278             mForecastAdapter.add(dayForecastStr);
279         }
280         // New data is back from the server. Hooray!
281     }
282 }
283 }
284 }

```

Then, go ahead and compile and build the app. When you run it, and you hit the refresh button, you should see a week's worth of weather data for your location. Once it's working, you can remove the verbose log statements so you don't clutter the logs.

From Review Material for Lesson 2:

HttpURLConnection

HttpURLConnection is a Java class used to send and receive data over the web. We use it to grab the JSON data from the [OpenWeatherMap API](#).

MainThread vs. Background Thread (Tao: good)

In Android there is a concept of the **Main Thread** or UI Thread. If you're not sure what a thread is in computer science, check out this [wikipedia article](#). The main thread is responsible for keeping the UI running smoothly and responding to user input. It can only execute one task at a time. If you start a process on the Main Thread which is very long, such as a complex calculation or loading process, this process will try to complete. While it is completing, though, your UI and responsiveness to user input will hang.

Therefore, whenever you need to start a longer process you should consider using another "Background" thread, which doesn't "block" the Main Thread. An easy (but by no means perfect) way to do this is to create a subclass of AsyncTask.

AsyncTask

AsyncTask is an easy to use Android class that allows you to do a task on a background thread

and thus not disrupt the Main Thread. To use AsyncTask you should subclass it as we've done with FetchWeatherTask. There are four important methods to override:

`onPreExecute` - This method is run on the UI before the task starts and is responsible for any setup that needs to be done.

`doInBackground` - This is the code for the actual task you want done off the main thread. It will be run on a background thread and not disrupt the UI.

`onProgressUpdate` - This is a method that is run on the UI thread and is meant for showing the progress of a task, such as animating a loading bar.

`onPostExecute` - This is a method that is run on the UI **after** the task is finished.

Note that when you start an AsyncTask, it is tied to the activity you start it in. When the activity is destroyed (which happens whenever the phone is rotated), the AsyncTask you started will refer to the destroyed activity and not the newly created activity. This is one of the reasons why using AsyncTask for a longer running task is dangerous.

Adding menu buttons

So that we could add a temporary Refresh button, we learned how to add menu buttons. Here are the basic steps.

1. Add an xml file in `res/menu/` that defines the buttons you are adding, their ordering and any other characteristics.

2. If the menu buttons are associated with a fragment, make sure to call `setHasOptionsMenu(true)` in the fragment's `onCreate` method.

3. Inflate the menu in the `onCreateOptionsMenu` with a line `inflater.inflate(R.menu.forecastfragment, menu);`

4. In `onOptionsItemSelected` you can check which item was selected and react appropriately. In the case of Refresh, this means creating and executing a `FetchWeatherTask`.

values/strings.xml

Android has a specific file for all of the strings in your app, stored in `values/strings.xml`. Why? Well besides further helping separate content from layout, the strings file also makes it easy to localize applications. You simply create a `values-language/strings.xml` files for each locale you want to localize to. For example, if you want to create a Japanese version of your app, you

would create a `values-ja/strings.xml` . Note, if you put the flag `translatable="false"` in your string it means the string need not be translated. This is useful when a string is a proper noun.

Permissions

By default, applications in Android are **sandboxed**. This means they have their own username, run on their own instance of the virtual machine, and manage their own personal and private files and memory. Therefore, to have applications interact with other applications or the phone, you must request permission to do so.

Permissions are declared in the `AndroidManifest.xml` and are needed for your app to do things like access the internet, send an SMS or look at the phone's contacts. When a user downloads your application, they will see all of the permissions you request. In the interest of not seeming suspicious, it's a good idea to only request the permissions you need.

JSON Parsing

Often when you request data from an API(API 的意思見下) this data is returned in a format like JSON. This is the case for Open Weather Map API. Once you have this JSON string, you need to parse it.

If you're unfamiliar with JSON, take a look at [this tutorial](#).

If you're not sure of the exact structure of the JSON, use a formatter. [Here's a good one](#) you can use in the browser.

In Android, you can use the `JSONObject` class, documented [here](#). To use this class you take your JSON string and create a new object:

```
JSONObject myJson = new JSONObject(myString);
```

And then you can use various `get` methods to extract data, such as `getJSONArray` and `getLong` .

From Wiki:

应用程序接口（[英语](#)：**A**pplication **P**rogramming **I**nterface，简称：**API**），又称为**应用编程接口**，就是**软件**系统不同组成部分衔接的约定。由于近年来软件的规模日益庞大，常常需要把复杂的系统划分成小的组成部分，编程接口的设计十分重要。程序设计的实践中，编程接口的设计首先要使软件系统的职责得到合理划分。良好的**接口**设计可以降低系统各部分的相互依赖，提高组成单元的**内聚性**，降低组成单元间的**耦合**程度，从而提高系统的维护性和扩展性。