CODE NOTES – CountryInfo_Student app

All quotes in this document from developer.android.com and https://developers.google.com/android/
No linked pages are required reading in this document

OVERVIEW

The main topics this week are Google Maps, Web Services, the WebView UI component, Executors and the JSON data format.

Google Maps is a set of API classes and interfaces grouped together in the com.google.android.gms.maps package. These are Google APIs not Android APIs. Basically, beyond a certain usage point you must pay to use them. The main class of the Google Maps API is GoogleMap. It is the entry point for all methods related to the displayed map.

To ensure the app is backwardly compatible as far as is possible the support version of several APIs have been used rather than their current Android framework equivalents.

Emulators struggle with fast dragging and pinching so manipulate the map slowly waiting for each operation to complete before starting the next one.

Web Services are like function calls made using HTTP (i.e. made over the Web). They are a way of CRUDing remote data. They used to involve (and many still do) rigorous messaging protocols and data formats. Modern Web Services use a RESTful paradigm which is much simpler. It uses the standard HTTP verbs GET, POST, PUT and DELETE to signal R, C, D, U data operations. It also employs a URL scheme such that each URL on the server represents a resource which can be a collection resource or an element resource. For example, to get the details about a country from a Web Service that offer such data we could make the following HTTP GET request: "https://restcountries.eu/rest/v2/name/mexico". You can try this now. Just paste the URL into your browser's address bar. The Web Service will send back data in JSON format. Your browser will realise this is not HTML so will not attempt to render a Web page but just dump the JSON data onto the screen (JSON is just text so this is possible).

WebView is just a View widget that displays Web pages.

> *"It does not include any features of a fully developed web browser, such as navigation controls or an address bar. All that WebView does, by default, is show a web page."*
>
> *"A common scenario in which using WebView is helpful is when you want to provide information in your application that you might need to update [frequently without editing, recompiling and resubmitting to app stores], such as an end-user agreement or a user guide. Within your Android application, you can create an Activity that contains a WebView, then use that to display your document that's hosted online."*

By making some settings and doing some coding a WebView can gain many of the features of a Web page rendered by a browser such as enabling JavaScript, page navigation and history, adding an address bar. WebView's offer very exciting native/mobile Web app hybrid capabilities.

Executor enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread.

JSON is an extremely lightweight data format popular for data transfer across the Web.

The app uses the following java classes:

MapsActivity (UI layout: activity_maps)

This is the launch Activity. Its UI layout is a single fragment element (id = map).

As usual when a Fragment is inserted using XML its element contains a name attribute (or you can use the class attribute) that specifies the Fragment class that will inflate its interface and implement its functionality. In this case this is not a developer coded Fragment class (as we have seen in previous weeks) but a Google class called com.google.android.gms.maps.SupportMapFragment. This class inflates the map image into the XML Fragment and then supports all the functionality we expect from a Google map.

Another component of the Google Maps API is an interface called OnMapReadyCallback. By implementing this interface, we promise to listen for and react to its only event (onMapReady) which signifies the map is ready for use. In the onMapReady event handler we can initialise the map as required and set up listeners to make the map interactive.

MapsActivity extends AppCompatActivity for maximum backward compatibility:

MapsActivity implements OnMapReadyCallback.

> *"Callback interface for when the map is ready to be used.*
>
> *Once an instance of this interface is set on a MapFragment or MapView object [or an Activity that contains one of these], the onMapReady(GoogleMap) method is triggered when the map is ready to be used and provides a non-null instance of GoogleMap."*

onCreate (Activity lifecycle callback)

A Geocoder instance is instantiated and its reference made available to the entire class by assigning it to a class level variable (btw Geocoder is an Android API not a Google API).

> *"A class for handling geocoding and reverse geocoding. Geocoding is the process of transforming a street address or other description of a location into a (latitude, longitude) coordinate. Reverse geocoding is the process of transforming a (latitude, longitude) coordinate into a (partial) address."*

A SupportMapFragment reference to the Activity's only Fragment, is also obtained. Note the use of getSupportFragmentManager not getFragmentManager as required by an app using the support library's Fragment class rather than the Android framework's Fragment class.

> *"public class SupportMapFragment extends Fragment*
>
> *A Map component in an app. This fragment is the simplest way to place a map in an application. It's a wrapper around a view of a map to automatically handle the necessary life cycle needs. Being a fragment, this component can be added to an activity's layout file simply [using XML]….. "*

The SupportMapFragment's getMapAsync method specifies which object will listen for an onMapReady event with its first and only parameter. The callback to handle this event is the only method of the OnMapReadyCallback interface. The current MapsActivity instance is specified as the listener object (i.e. the parameter is **this**).

The MapsActivity class therefore implements the OnMapReadyCallback interface which includes a single method header (onMapReady) in which a response to the map being ready can be handled. A reference to the map is passed into the onMapReady event handler so the map can be programmatically accessed and manipulated.

onMapReady (OnMapReadyCallback interface method)

The map is now ready for use and we have a GoogleMap reference to it (input parameter of onMapReady) so all the functionality of a GoogleMap can be accessed.

Some initialisation of the GoogleMap is performed including setting up a listener containing an event handler (onMapClick) to handle clicks on the map. The listener is set up using the usual compressed syntax.

The onMapClick event handler's input parameter is a LatLng object containing the latitude and longitude of the click point on the map, The coded response to a click on the GoogleMap instance is:

- Use the Geocoder instance's getFromLocation method to translate the passed in LatLng value to an ArrayList (addresses) of address objects returned by the Geocoder instance. In all my debugging experiments the ArrayList only contains one address object.
- Get the country name from the address object at index 0 using its class's getCountryName method
- Pop up a Snackbar asking if more details about the country are required
    - The Snackbar has a conditional action (depending on whether there was an address object in the addresses ArrayList)
        - The setAction method of the Snackbar uses a ternary conditional operator (see here at the end of The Conditional Operators section) to compress syntax.
    - The action's click listener can be any object whose class implements the View.OnClickListener interface
    - A nested class is set up to implement this interface + its constructor is set up to have the selected country's name passed in
        - The promised onClick method starts an intent to the CountryDetails Activity after inserting the selected country's name in the intent's extras Bundle

ActionClickListener (Nested Class of MapsActivity)

See discussion of Snackbar's action immediately above.

Java Checked Exceptions

The call to the Geocoder's getFromLocation can generate a IOException among other exceptions (see here). An IOException is a sub class of Exception but not of RuntimeException which makes it a Java **checked** exception by definition. This means the call to getFromLocation must be enclosed in a try/catch block or it's a syntax error. The usefulness of Java checked exceptions is hotly debated amongst Java developers.

continued …

CountryDetails (UI layout: activity_country_details)

The layout is just a bunch of TextViews, half of them serving as labels (text set statically in XML) for the other half which display some important data about the selected country retrieved using a Web Service (text set dynamically in code). This country was selected in the MapsActivity Activity and its name passed in an Intent to the CountryDetails Activity from that Activity.

onCreate (Activity lifecycle callback)

Inflates the Activity's UI layout layout then sets up a home button in its app bar (setDisplayHomeAsUpEnabled(true)). The manifest must also be modified to set up the app's parent/child Activity hierarchy (android:parentActivityName) to make the home button work correctly.

The selected country name is extracted from the extras Bundle of the intent that launched this Activity. TextView references are captured using findViewById.

Executors Class

We are going to be retrieving data across the Web using a Web Service. How long will it take? Unknown, so we better NOT do the retrieving on the main UI thread else we'll lock up the UI. But that involves the Thread and Handler classes and it all gets a bit complicated. Executors to the rescue.

In order to have multi-threaded applications that runs some tasks in the background and update the UI, we must have at least two pointers/references, the first pointer references the background thread and the second one points at the UI thread. Now, lets create the first one:


```
ExecutorService executor = Executors.newSingleThreadExecutor();
```


In the statement above, we create a reference to a single threaded executor. The executor that is going to run our tasks in the background.

> *If you application requires more than on thread (multiple concurrent tasks) then you can create a pool of threads (see HERE)*

Now, let's create the second pointer which is a reference to a hanlder that uses the UI thread to update the UI elements.

```
Handler uiHandler=new Handler(Looper.getMainLooper());
```

Now, the handler will be used to publish the results of the executor (see above) on the UI thread.

Let's execute a task

To send/submit a task to the executor to run it in the background thread, we must call the submit method as shown below.

```
executor.execute(() -> {
//your task goes here

//Now to update the UI elements
uiHandler.post(()->{

  }
});
```


HTTP Requests

The Web Service is what is known as RESTFul. One aspect of such Web Services is that "Each URL on the server represents a resource; either a collection resource or an element resource."

In this case we have base URL + "/" + country name (https://restcountries.eu/rest/v2/name/mexico). You can use this in any browser. Try it to see what the JSON formatted data looks like. It looks like this:

[{"name":"Mexico","topLevelDomain":[".mx"],"alpha2Code":"MX","alpha3Code":"MEX","callingCodes":["52"],"capital":"Mexico City","altSpellings":["MX","Mexicanos","United Mexican States","Estados Unidos Mexicanos"],"region":"Americas","population":122273473,"latlng":[23.0,-102.0],"demonym":"Mexican","area":1964375.0,"gini":47.0,"timezones":["UTC-08:00","UTC-07:00","UTC-06:00"],"borders":["BLZ","GTM","USA"],"nativeName":"México","numericCode":"484","currencies":[{"code":"MXN","name":"Mexican peso","symbol":"$"}],"languages":[{"iso639_1":"es","iso639_2":"spa","name":"Spanish","nativeName":"Español"}],"translations":{"de":"Mexiko","es":"México","fr":"Mexique","ja":"メキシコ","it":"Messico","br":"México","pt":"México"},"flag":"https://restcountries.eu/data/mex.svg"}]

Don't panic it's actually a very simple format, we will discuss it below. There is a neatly formatted dump of this JSON at the end of these code notes.

The following procedure is used to get JSON formatted data from the Web Service:

- A try block is required as several checked exceptions can be thrown while attempting to call a Web Service
- Use a String to create the required URL
- Get an HttpsURLConnection object representing a connection to the URL's server
    - The connection object contains a response code indicating the success/failure of the connection
- If the connection is ok open a data "pipe" to the server (InputStream) and put a byte to character translator in the pipe (InputStreamReader)
- Now connect the pipe to a JsonReader
- Now apply methods to the JsonReader to get it to read as many characters down the pipe as required to form the next JSON token (i.e. meaningful bit of JSON)
    - This is done before during and after 2 nested while loops
        - The outer loop processes array objects
            - The Web Service will return an array object for each country with the supplied

name anywhere in its name e.g. india returns an object for “India” but also one for “British Indian Ocean Territory”

- Hence the outer loop's condition

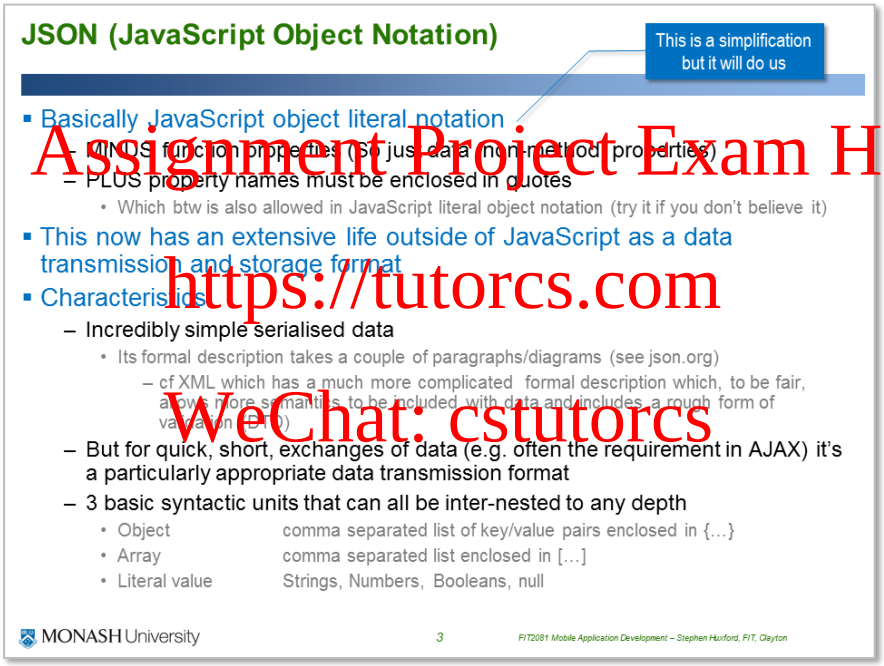
- The inner loop processes properties inside each object
  - The selection structure looks a bit of a mess but remember you are constrained by parsing Java sequentially token by token

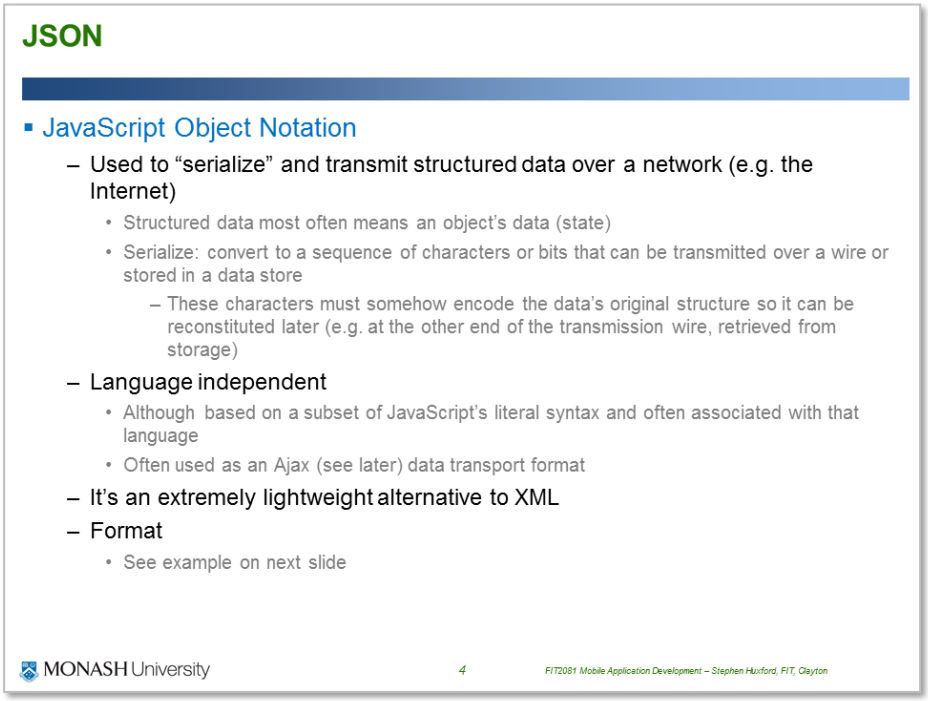
- Capture the data tokens (rather than the JSON syntax tokens) as required and use them to populate a CountryInfo object (It’s a straightforward class nested in the CountryDetails class that hold all the country info we are interested in)

CountryInfo (Nested class)

a straightforward class nested in the CountryDetails class that holds all the country info we are interested in

JSON





JSON (cont.)

JSON - Example

property

value

objects can be nested

array of objects

array elements can be objects

this is a valid JavaScript object literal AND a valid JSON object

(the former allows quoted property names the latter insists on them)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumbers": [
        { "type": "home", "number": "212 555-1234" },
        { "type": "fax", "number": "646 555-4567" }
    ],
    "newSubscription": false,
    "companyName": null
}
```

http://en.wikipedia.org/wiki/JSON

JSON's basic types are: http://en.wikipedia.org/wiki/JSON

Number	(integer, real, or floating point)
String	(double-quoted Unicode with backslash escaping)
Boolean	(true and false)
Array	(an ordered sequence of values, comma-separated and enclosed in square brackets)
Object	(collection of key:value pairs, comma-separated and enclosed in curly braces)
null	

JSON

This slide's text mostly copied from:
http://www.json.org/js.html (http://www.json.org is an excellent JSON resource)

"

- JSON is a Subset of the Object Literal Notation of JavaScript
  - Therefore the following is valid

```
var myJSONObject =
{"bindings": [
{"ircEvent": "PRIVMSG", "method": "newURI", "regex": "^http://.*"},
{"ircEvent": "PRIVMSG", "method": "deleteURI", "regex": "^delete.*"},
{"ircEvent": "PRIVMSG", "method": "randomURI", "regex": "^random.*"}
] };
```

    - In this example, an object is created containing a single property "bindings" which contains an array containing three objects, each containing "ircEvent", "method", and "regex" properties.
- To convert a JSON String into a JS Object
  - you can use the eval() function.
  - eval() invokes the JavaScript compiler. Since JSON is a proper subset of JavaScript, the compiler will correctly parse the string and produce an object structure.
  - The text must be wrapped in parenthesis to avoid tripping on an ambiguity in JavaScript's syntax.
    - var myJSONObject = eval('(' + myJSONstring + ')');
- Object Members (Properties) can then be Retrieved Using Dot and/or Subscript Operators
  - e.g. myJSONObject.bindings[0].method // "newURI"
- The eval Function is Very Fast. However, it can Compile and Execute any JavaScript Program, So there can be Security Issues.
  - The use of eval is indicated when the source is trusted and competent. [Otherwise] It is much safer to use a JSON parser [such as the parseJSON() method from json.js].

"

JSON message received in response to https://restcountries.eu/rest/v2/name/mexico


```
[
    {
        "name":"Mexico",      "topLevelDomain":[".mx"],
        "alpha2Code":"MX",
        "alpha3Code":"MEX",
        "callingCodes":["52"],
        "capital":"Mexico City",
        "altSpellings":["MX", "Mexicanos", "United Mexican States", "Estados Unidos Mexicanos"],
        "region":"Americas",
        "population":122273473,
        "latlng":[23.0, -102.0],
        "demonym":"Mexican","area":1964375.0,
        "gini":47.0,
        "timezones":["UTC-08:00", "UTC-07:00", "UTC-06:00"],
        "borders":["BLZ", "GTM", "USA"],
        "nativeName":"México",
        "numericCode":"484",
        "currencies":[{"code":"MXN", "name":"Mexican peso", "symbol":"$"}],
        "languages":[{"iso639_1":"es", "iso639_2":"spa", "name":"Spanish", "nativeName":"Español"}],
        "translations":{"de":"Mexiko",   "es":"México",   "fr":"Mexique",   "ja":" メ キ シ コ ",  "it":"Messico",   "br":"México",
                        "pt":"México"},
        "flag":"https://restcountries.eu/data/mex.svg"
    }
]
```


So in this case, one array with a single object element containing 21 property name/value pairs. Some of these values are themselves arrays or arrays of objects or objects.