Advanced Mobile Application Development
Week 3: External Data

Network Connections

https://developer.android.com/training/basics/network-ops

Each Android application runs in its own process in a single thread which is called the **Main thread** or **UI thread.** Prior to Ice Cream Sandwich(API 14/15), Android apps could perform any operations from the main UI thread. This led to such situations where the application would be completely unresponsive while waiting for a response. Now there are stricter rules regarding performing operations -- tasks have 100-200ms to complete a task in an event handler. Operations that take longer could be killed by the WindowManager or ActivityManager processes and the user will receive an application not responding(ANR) message.
In scenarios where we're not in control of how long the operation will take we shouldn't perform them on the main thread. Instead we should perform them on a thread that runs in the background so our main UI thread never freezes or errors out.

- File IO
- Database interaction
- Network interaction
- API integration

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

AsyncTask

https://developer.android.com/reference/android/os/AsyncTask.html
The **AsyncTask** class enables you to implement a background thread to perform operations asynchronously and not hold up the UI thread. The results of the background thread can then be published on the UI thread without having to manipulate threads and/or handlers. The AsyncTask class should ideally be used for short operations, a few seconds at the most. Longer operations should use a service. The AsyncTask must be subclassed to be used.

AsyncTask types

An asynchronous task is defined by 3 generic types:

1. **Params**, the type of the parameters sent to the task
2. **Progress**, the progress type published during the task
3. **Result**, the type of the result returned on completion of the task.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type Void.

AsyncTask's 4 steps

The AsyncTask subclass is automatically loaded on the UI thread and the task instance must be created on the UI thread. When an asynchronous task is executed, the task goes through 4 steps:

1. **onPreExecute(),** invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. **doInBackground(Params...),** invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take longer than 200ms. The parameters of the asynchronous task are passed to this step. The

result of the computation must be returned by this step and will be passed back to the last step. The AsyncTask class manages this background thread.
3. Inside the **doInBackground(Params...)** method you can call **publishProgress(Progress...)** to display any form of progress. This will automatically call **onProgressUpdate(Progress...),** which is invoked on the UI thread to display the progress. For instance, it can be used to animate a progress bar or show logs in a text field.
4. **onPostExecute(Result),** invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Network Request
Android and Java have classes that enable you to download JSON files before parsing them.

The **HttpURLConnection** class supports sending and receiving data over HTTP. Thee openConnection() method to make a connection to a given URL.

The **InputStream** class gives us access to any stream of data. A stream is basically any set of characters. You can use the getInputStream() method to get the input stream for our JSON file. InputStream works only on bytes so we use the **InputStreamReader** class to convert a byte stream into a character stream which the **BufferedReader** class works with. The readLine() method reads one line at a time from the BufferedReader character stream. We can then convert the character stream to a String using the **StringBuilder** class which creates a mutable sequence of characters. We create a String from this sequence using the toString() method.

Retrofit and Volley both are the REST client libraries that will handle all of this networking for us.

Volley
https://developer.android.com/training/volley
Volley is an HTTP library that makes networking for Android apps easier and faster.
Volley offers the following benefits:

- Automatic scheduling of network requests
- Multiple concurrent network connections
- Disk and memory response caching
- Support for request prioritization.
- Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
- Ease of customization, for example, for retry and backoff.
- Strong ordering that makes it easy to correctly populate your UI with data fetched asynchronously from the network.
- Debugging and tracing tools.

Volley uses a `RequestQueue` that manages worker threads for running the network operations, reading from and writing to the cache, and parsing responses. You can use these Volley convenience methods or create a custom RequestQueue.

- The `Volley.newRequestQueue` method sets up a `RequestQueue` using default values and starts the queue
- To send a request you construct a request object and add it to the `RequestQueue` using the `add(stringRequest)` method

- Once you add the request it moves through the pipeline, gets serviced, and has its raw response parsed and delivered.

Volley supports these request types or you can create a custom request

- StringRequest -- receives a raw string from a URL
- JsonObjectRequest (subclass of JsonRequest) – receive a JSON object from a URL
- JsonArrayRequest (subclass of JsonRequest) – receive a JSON array from a URL

Request objects do the parsing of raw responses and Volley takes care of dispatching the parsed response back to the main thread for delivery.

More info on how Volley handles requests https://developer.android.com/training/volley/simple#send

If your application makes constant use of the network, it's probably most efficient to set up a single instance of `RequestQueue` that will last the lifetime of your app. You can achieve this by implementing a singleton class that encapsulates `RequestQueue` and other Volley functionality. This is more modular and the preferred approach over subclassing Application and setting up the queue in `Application.onCreate()`.

A key concept is that the `RequestQueue` must be instantiated with the `Application` context, not an `Activity` context. This ensures that the `RequestQueue` will last for the lifetime of your app, instead of being recreated every time the activity is recreated (for example, when the user rotates the device).

Volley is not suitable for large download or streaming operations, since Volley holds all responses in memory during parsing. For large download operations, consider using an alternative like `DownloadManager`.

Add the following dependency to your app's build.gradle file:

```
implementation 'com.android.volley:volley:1.1.1'
```

API

I created an app that uses the Movie Database (TMDb) API
https://developers.themoviedb.org/3/getting-started/introduction

Sign up https://www.themoviedb.org/account/signup

Documentation/Getting started https://developers.themoviedb.org/3/getting-started/authentication

Register for an API key, click the API link from within your account settings page.
https://www.themoviedb.org/settings/api/request individual developer, accept the agreement.
Al fields are required.

Now you have an API key and a sample URL
https://api.themoviedb.org/3/movie/550?api_key=9bc41deb95194da5e8865be1fe7750a4

We'll be using the top rated movies API https://developers.themoviedb.org/3/movies/get-top-rated-movies
Use the try it out tab to send a sample request and see the JSON response. You can also do this directly in your browser.

When using an API you have to look at the JSON to understand the structure of the data. Each API will have a different structure. This one is pretty straight forward.
Keys: page, total_results, total_pages, and results. The value for results is an array. Top rated returns 20 results, which is plenty for the sample app.
Look at one of the results to see what data is sent back.

The next step is to figure out what data we want to use from the API.
id: value is an int
title: value is a String
vote_average: value is a Double
poster_path: value is the name of the jpg (note it is not the full path)

Movies
Create a new Android project called Movies with a minimum API of 21 using the Empty Activity template.

Our application requires internet access, so we must request the INTERNET permission in the AndroidManifest.xml file. Add the following before the application tag.

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

Add the dependencies for Material Components, RecyclerView, Volley, CardView and Picasso into the app's grade file.
```
implementation 'com.google.android.material:material:1.1.0-rc01'
implementation 'androidx.recyclerview:recyclerview:1.1.0'
implementation 'com.android.volley:volley:1.1.1'
implementation 'androidx.cardview:cardview:1.0.0'
implementation 'com.squareup.picasso:picasso:2.71828'
```

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

Layout
In the layout file delete the default textview.
With Autoconnect on in design view drag out a RecyclerView to display the list(can be found in common or containers).
Give the RecyclerView an id `android:id="@+id/recyclerView"`
Add start, end, top, and bottom constraints of 0.
Make sure the RecyclerView has layout_width and layout_height set to "match_constraint" (0dp).

List Item Layout
We also need a layout file to determine what each row in our list will look like. I decided to use a cardView.
File | New | Layout resource file
File name: card_list_item
Root element: androidx.cardview.widget.CardView

Source set: main
Directory name: layout

Change the constraint layout's height to "wrap_content". If the height is match_constraint each text view will have the height of a whole screen.

```
android:layout_height="wrap_content"
```

Then I added a constraint layout as a child of the CardView and defined each row in that.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```

For each movie we'll show the poster in an ImageView, the title in a TextView and the vote average in a TextView. Add those views, give them each ids, constraints, and any padding or styling you want. Once you don't need the default text to help with layout, remove it.

Java class

We're going to create a custom Java class to represent the data we'll be using.
First create a new package for our model. In the java folder select the tulips folder (not androidTest or test)
File | New | Package
Name: model

Then select the new model package and add a new class called Bulb.
File | New | Java class
Name: Movie
Kind: Class

The Movie class will store the movie id, title, posterURL, and rating. Instead of saving the vote average as a Double and converting it to a String every time I needed to display it in a TextView, I decided to save it as a String and I'll just do the conversion once. This works since I won't be doing any mathematical operations on it, I will always need it as a String.
We'll have a constructor that takes all these as arguments and getter and setter methods for them.
For the constructor, getter, and setter methods Android Studio can generate the code for you.
Code | Generate | Constructor (pick whichever data members you want as arguments, we want both)
Code | Getter and Setter

```
public class Movie {

    private int id;
    private String title;
    private String posterURL;
    private String rating;

    public Movie(int id, String title, String posterURL, String rating) {
        this.id = id;
        this.title = title;
        this.posterURL = posterURL;
        this.rating = rating;
    }

    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getPosterURL() {
        return posterURL;
    }

    public void setPosterURL(String posterURL) {
        this.posterURL = posterURL;
    }

    public String getRating() {
        return rating;
    }

    public void setRating(String rating) {
        this.rating = rating;
    }
}
```

When we download the data from the API we're going to need a way to let our Activity know when the data is done being downloaded. To do that we're going to define an interface.

Create another package called loader and then a Java class in loader called OnDataReadyCallback
File | New | Java class
Name: OnDataReadyCallback
Kind: Interface

The interface will just have one method that will take the list of movies.

```
public interface OnDataReadyCallback {
    void onDataReady(List<Movie> movieList);
}
```

Create another Java class in loader called JSONData where we will request the data from the API and parse the JSON data.
File | New | Package
Name: loader

Then select the new model package and add a new class called JSONData.
File | New | Java class
Name: JSONData

Kind: Class

```
    public static void getJSON(Context context, final OnDataReadyCallback
callback){
        String url = API_URL + API_KEY;

        // create Volley request queue
        RequestQueue queue = Volley.newRequestQueue(context);

        StringRequest stringRequest = new StringRequest(Request.Method.GET,
url,
                new Response.Listener<String>() {
                    @Override
                    public void onResponse(String response) {
                        movieList = parseJSON(response);
                        callback.onDataReady(movieList);
                    }
                }, new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                Log.e("ERROR", error.getMessage(), error);
            }
        });
        queue.add(stringRequest);
    }

    private static List<Movie> parseJSON(String response){
        // Base url for the posters
        final String POSTER_BASE_URL = "https://image.tmdb.org/t/p/w185";

        if (response != null) {
            try {
                //create JSONObject
                JSONObject jsonObject = new JSONObject(response);

                //create JSONArray with the value from the characters key
                JSONArray resultsArray = jsonObject.getJSONArray("results");

                List<Movie> movieData = new ArrayList();
                //loop through each object in the array
                for (int i =0; i < resultsArray.length(); i++) {
                    JSONObject movieObject = resultsArray.getJSONObject(i);

                    //get values
                    int id = movieObject.getInt("id");
                    String title = movieObject.getString("title");
                    //save the fully qualified URL for the poster image
                    String posterURL = POSTER_BASE_URL +
movieObject.getString("poster_path");
                    String rating =
String.valueOf(movieObject.getDouble("vote_average"));

                    //create new Movie object
```

```
                    Movie movie = new Movie(id, title, posterURL, rating);

                    //add movie object to our ArrayList
                    movieData.add(movie);
                }
                return movieData;
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

Adapter

Now we need to create an adapter which will be used to populate the data into the RecyclerView. The adapter's role is to convert an object at a given position into a list row item to be inserted in the list.

In the java folder create a new class for our adapter.
File | New | Class
Name: MyListAdapter
Superclass: androidx.recyclerview.widget.RecyclerView.Adapter

With a RecyclerView the adapter requires the existence of a ViewHolder object which describes and provides access to all the views within each item row.
To indicate that our class will be implementing the ViewHolder class add it to the class definition.

```
public class MyListAdapter extends RecyclerView.Adapter<
MyListAdapter.ViewHolder>
```

You will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.
Every adapter has three required methods that we need to implement.
You will still have an error because we need to define the ViewHolder class which will set up the bindings to the view in the layout file.
In our MyListAdapter class we'll create a ViewHolder class to render the item and a default constructor.

```
public class MyListAdapter extends
RecyclerView.Adapter<MyListAdapter.ViewHolder> {
    class ViewHolder extends RecyclerView.ViewHolder{
        TextView mTextView;
        TextView mTextView2;
        ImageView mImageView;

        public ViewHolder(@NonNull View itemView) {
            super(itemView);
            mTextView = itemView.findViewById(R.id.textView);
            mTextView2 = itemView.findViewById(R.id.textView2);
            mImageView = itemView.findViewById(R.id.imageView);
        }
    }
```

This should have fixed the errors.

With the viewholder defined, let's set up the MyListAdapter class. We'll define a list of movies, Context and a constructor that takes both. We'll use this later in MainActivity to define an instance of this class.

```
    private List<Movie> mMovieList;
    private Context mContext;;

    public MyListAdapter(List<Movie> mMovieList, Context mContext) {
        this.mMovieList = mMovieList;
        this.mContext = mContext;
    }
```

Now we'll implement the 3 required methods.
OnCreateViewHolder() is called automatically by the adapter each time it needs to display a data item. When you inflate the xml layout file, you get a view, and then you wrap that in an instance of your ViewHolder class and return that object.

```
@NonNull
@Override
public MyListAdapter.ViewHolder onCreateViewHolder(@NonNull ViewGroup
parent, int viewType) {
    LayoutInflater inflater = LayoutInflater.from(mContext);
    View itemView = inflater.inflate(R.layout.card_list_item, parent,
false);
    ViewHolder viewHolder = new ViewHolder(itemView);
    return viewHolder;
}
```

OnBindViewHolder is called each time the adapter needs to display a new data item. It passes in the reference to the ViewHolder and the position of the data item in the collection. The job of OnBindViewHolder is to take that data object and display its values.

We'll also need a method that we can call when there's new data and let the adapter know the data has changed.

```
public void setMovieList(List<Movie> movieList){
    this.mMovieList = movieList;
    notifyDataSetChanged();
}
```

Picasso
https://square.github.io/picasso/
Downloading the poster image introduces some issues around both the network request and the caching of the images. The Picasso library handles many common pitfalls of image loading on Android:

- Handling ImageView recycling and download cancelation in an adapter
- Complex image transformations with minimal memory use
- Automatic memory and disk caching

It also lets us easily define placeholder and error images (note that these are not automatically resized, so do that before you add them to your project. Mine ended up 75x56).
In the newest version of Picasso, 2.71828, the with(context) method is deprecated and is replaced with get() method.
https://github.com/codepath/android_guides/wiki/Displaying-Images-with-the-Picasso-Library

```
@Override
public void onBindViewHolder(@NonNull MyListAdapter.ViewHolder holder, int
position) {
    Movie movie = mMovieList.get(position);
    holder.mTextView.setText(movie.getTitle());
    holder.mTextView2.setText("Rating: " + movie.getRating());
    Picasso.get().load(movie.getPosterURL())
            .error(R.drawable.image_placeholder)
            .placeholder(R.drawable.image_placeholder)
            .into(holder.mImageView);
}
```

getItemCount() returns the number of items in the collection.

```
    @Override
    public int getItemCount() {
        return mMovieList.size();
    }
```

MainActivity

First I'll get access to the array list of movies from my JSONData class to use in this class and define the adapter.

With the adapter set up, in onCreate() we need to instantiate an adapter with our sample data.
Then we need to set the adapter to the RecyclerView.

We also need to set a Layout Manager for our RecyclerView instance. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.

We're using a LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically).

Our MainActivity will implement the OnDataReadyCallback interface and implement its onDataReady method where we'll pass the updated movie list to the adapter using its setMovieList() method which also updates the adapter so the data is shown.

```
public class MainActivity extends AppCompatActivity implements
OnDataReadyCallback {
    List<Movie> topMovieList =JSONData.movieList;
    private MyListAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //get data
        if (topMovieList.isEmpty()) {
            JSONData.getJSON(this, this);
```

```
        }

        //get the recycler view
        RecyclerView recyclerView = findViewById(R.id.recyclerView);

        //define an adapter
        MyListAdapter adapter = new MyListAdapter(topMovieList, this);

        //assign the adapter to the recycler view
        recyclerView.setAdapter(adapter);

        //set a layout manager on the recycler view
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
    }

     @Override
     public void onDataReady(List<Movie> movieList) {
         adapter.setMovieList(movieList);
     }

}
```

Your app should now be able to request the JSON data from the API, parse it and display it in the RecyclerView.
You could build on this and have the card expand when the user clicks on it to show the movie overview or other information available through the API.