Advanced Mobile Application Development
Week 12: Android and JSON

Keeping data externalized is better than hard coding data into your app. Data is often incorporated into your app from a variety of sources in different formats. Outside of SQL databases the most common data formats are XML and JSON.

Android provides classes to manipulate JSON data

- **JSONObject** https://developer.android.com/reference/org/json/JSONObject
  - Represents a JSON object as a modifiable set of name/value mappings.
    - Names are unique, non-null strings
    - Values may be any mix of `JSONObject`, `JSONArray`, Strings, Booleans, Integers, Longs, Doubles or `NULL`.
  - Many different get methods to access different types of data
    - getJSONObject() returns a JSON object
      - { } represents a JSON object
    - getJSONArray() returns an array
      - [ ] represents a JSON array
- **JSONArray** https://developer.android.com/reference/org/json/JSONArray
  - Represents a JSON array which can be the value for a given key
    - Values may be any mix of `JSONObject`, other `JSONArray`, Strings, Booleans, Integers, Longs, Doubles, or null
  - Many different get methods to access different types of data
- **JSONException**
  - handles all the exceptions related to JSON parsing.

There are JSON parsing libraries as well:

- Google's Gson library for converting between JSON and Java objects. Gson will also automatically map JSON keys into your model class. This helps to avoid needing to write boilerplate code to parse JSON responses yourself. https://github.com/google/gson/blob/master/UserGuide.md
- Jackson is a suite of data-processing tools for Java including the JSON parser/generator library, and matching data-binding library (POJOs to and from JSON) https://github.com/FasterXML/jackson
- Moshi is another JSON library for Android that makes it easy to parse JSON into Java objects and serialize Java objects as JSON. Moshi also includes Kotlin support. https://github.com/square/moshi

Resources

https://developer.android.com/guide/topics/resources/providing-resources

We've already used many different types of resources in Android Studio such as values, layouts, and drawables.

This is also where you would include any static data files that your app will use.

By externalizing your app resources, you can access them using resource IDs that are generated in the project's `R` class.
There are many directories supported in your resources directory including the following for data:

- raw – used for files in their raw form including JSON.
    - To open these resources call resources.openRawResource(R.raw.*filename*)
        - returns an instance of the InputStream class
    - InputStream is an abstract class that you can use to manipulate the bytes in an ordered stream of bytes such as a JSON file
        - bufferedReader() creates a buffered reader on the InputStream
        - readText() reads the input stream completely as a String
        - other methods available for manipulating and reading files using the BufferedReader class
- assets – used for files if you need access to original file names and file hierarchy
    - files in `assets/` aren't given a resource ID, so you can read them only using AssetManager.
- xml – used for XML files
    - can be read at runtime by calling getResources().getXml(R.xml.*filename*)
        - returns a XMLResourceParser which is a subclass of XMLPullParser
        - more efficient than adding it as an asset and opening it as an input stream

Harry Potter
Create a new project
Empty Activity template
Name: Potter
Package name: the fully qualified name for the project
Language: Kotlin
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

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

Layout
Add any images you'll be using in your layout such as hogwarts.png into the drawable folder.
In activity _main.xml make sure Autoconnect (magnet) is turned on.
Remove the textview and add an ImageView to the top of the view and select the hogwarts image.

```
android:src="@drawable/hogwarts"
```

Add any missing constraints. If you want it with no gap/border add

```
android:adjustViewBounds="true"
```

Having a content description for an image is optional but makes your app more accessible.

In your strings.xml add
```
<string name="description">Header image</string>
```
In activity_main.xml use it for the contentDescription
```
android:contentDescription="@string/description"
```

In the layout file in design view drag out a RecyclerView to display the list (can be found in common or containers) below the image.

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) and the top constraint is to the imageView and not parent.

List Item Layout
We also need a layout file to determine what each row in our list will look like. Ours will be super simple with just a TextView.
File | New | Layout resource file
File name: list_item
Root element: androidx.constraintlayout.widget.ConstraintLayout
Source set: main
Directory name: layout

Drag out a TextView and make sure it has an id (textView)
Make sure the TextView has layout_width and layout_height set to "wrap_content".
Add start and top constraints of 0.
I also added some top and bottom padding so the text is in the vertical center of the row, some start padding, and made the text larger.
```
android:textAppearance="@style/TextAppearance.MaterialComponents.Headline6"
android:paddingBottom="5dp"
android:paddingTop="5dp"
android:paddingStart="10dp"
```

Once you don't need the default text to help with layout, remove it.

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.

JSON
We're going to add our json file as a raw resource.
Select the res folder and add a new Android Resource directory.
Directory name: raw
Resource type: raw
Source Set: main

Then drag or copy/paste harrypotter.json into the raw directory.

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

Then select the new model package and add a new class called PotterCharacter.
File | New | Kotlin class
Name: PotterCharacter
Kind: Class

The PotterCharacter class will be a data class that will store the character name and a String for the URL (info) that will be defined in the primary constructor in the class header.

```
data class PotterCharacter(val name: String, val info: String){}
```

Now create another package in potter called data and then a Kotlin class in data called JSONdata where we will load and parse our JSON data.
File | New | Package
Name: data

Then select the new data package and add a new class called JSONdata.
File | New | Kotlin class
Name: JSONdata
Kind: Class

In this class we're creating an ArrayList to hold our data and methods that will load the JSON from the resource file, parse the JSON, and return the ArrayList so we can use it in our MainActivity class.

```
class JSONdata() {
    var characterList = ArrayList<PotterCharacter>()

    fun getJSON(context: Context): ArrayList<PotterCharacter>{
        var json = loadJSONFromRes(context)
        characterList = parseJSON(json)
        return characterList
    }

fun loadJSONFromRes(context: Context): String{
    //opens the raw JSON file and assigns it to an InputStream
```

```
instance
    val inputStream =
context.resources.openRawResource(R.raw.harrypotter)
    //creates a buffered reader on the InputStream and readText()
returns a String
    val jsonString = inputStream.bufferedReader().use {it.readText()}
    return jsonString
}


fun parseJSON(jsonString: String): ArrayList<PotterCharacter>{
    try {
        //create JSONObject
        val jsonObject = JSONObject(jsonString)

        //create JSONArray with the value from the characters key
        val characterArray = jsonObject.getJSONArray("characters")

        //loop through each object in the array
        for (i in 0 until characterArray.length()){
            val item = characterArray.getJSONObject(i)

            //get values for name and info keys
            val name = item.getString("name")
            val info = item.getString("info")

            //create new PotterCharacter object
            val newCharacter = PotterCharacter(name, info)

            //add character object to our ArrayList
            characterList.add(newCharacter)
        }
    } catch (e: JSONException) {
        e.printStackTrace()
    }
    return characterList
}
}
```

In Kotlin we can usually use a simple for-in loop:
`for (item in collection)`
But if you try that with an ArrayList you get the error:
`for-loop range must have an iterator() method`
That's because the JSONArray class does not expose an iterator. So you have to iterate through a JSONArray using an index range which is 0 until the length of the array.

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/potter folder create a new class for our adapter.
File | New | Kotlin Class
Name: MyListAdapter

The class will need to subclass RecyclerView.Adapter

```
class MyListAdapter: RecyclerView.Adapter {}
```

AS should add the import for the RecyclerView class:

```
import androidx.recyclerview.widget.RecyclerView
```

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.

```
class MyListAdapter: RecyclerView.Adapter<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.

```
class ViewHolder(view: View): RecyclerView.ViewHolder(view) {
    val itemTextView: TextView = view.findViewById(R.id.textView)
}
```

This should have fixed the errors.
With the viewholder defined, let's set up the MyListAdapter class. We'll define a list of characters and add it to the primary constructor.

```
class MyListAdapter(private val characterList:
ArrayList<PotterCharacter>): RecyclerView.Adapter<ViewHolder>() {}
```

Make sure the import for your ViewHolder class was added, otherwise you'll need to specify MyListAdapter.ViewHolder.

```
import com.example.potter.MyListAdapter.ViewHolder
```

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.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
MyListAdapter.ViewHolder {
    //create an instance of LayoutInflater
    val layoutInflater = LayoutInflater.from(parent.context)
    //inflate the view
    val itemViewHolder = layoutInflater.inflate(R.layout.list_item,
parent, false)
    return ViewHolder(itemViewHolder)
}
```

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.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    //get data at the position
    val item = characterList[position]
    //set the text of the textview to the name
    holder.itemTextView.text = item.name
}
```

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

```
override fun getItemCount() = characterList.size
```

Go into MainActivity.kt.
We need to define an ArrayList that will hold our data and we'll do that at the class level.

In onCreate() we need to

1. populate our ArrayList with the JSON data
2. set up our recyclerView
3. instantiate an adapter with our sample data
4. set the adapter to the RecyclerView
5. set a Layout Manager for our RecyclerView instance
    a. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.
    b. We're using a LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically).

```
class MainActivity : AppCompatActivity() {
    var potterList = ArrayList<PotterCharacter>()
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        if (potterList.isEmpty()){
            val data = JSONdata()
            //populate with JSON data
            potterList = data.getJSON(this)
        }
        //get the recycler view
        val recyclerView: RecyclerView =
findViewById(R.id.recyclerView)

        //divider line between rows
        recyclerView.addItemDecoration(DividerItemDecoration(this,
LinearLayoutManager.VERTICAL))

        //define an adapter
        val adapter = MyListAdapter(potterList)

        //assign the adapter to the recycle view
        recyclerView.adapter = adapter

        //set a layout manager on the recycler view
        recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)

    }
}
```

You should be able to run it and see your list of characters.

Click Handler

To attach a click handler to items in our MyListAdapter class we'll add a second parameter in the constructor that represents the click listener. This parameter will actually be a function. This function takes an instance of our class PotterCharacter as a parameter and since it doesn't return anything the return type is Unit. Because this is passed into the class it's available throughout the class.

```
class MyListAdapter(private val characterList:
ArrayList<PotterCharacter>, private val
clickListener:(PotterCharacter) -> Unit):
RecyclerView.Adapter<ViewHolder>() {}
```

We want each item in the list to be able to handle click events so we need to set a click event listener to each view holder. The easiest place to do this is in onBindViewHolder() since we have access to the specific item. So now every view holder will have a click event as well.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    ...
    //assign click listener
    holder.itemView.setOnClickListener{clickListener(item)}
}
```

Now MyListAdapter and ViewHolder are ready to process click events.
To handle click events on list items we need to define what we want to do when a click event fires.
In MainActivity we'll create a method to define what should happen when an item is clicked.
When an item is clicked we want to open the url in a web browser.

```
private fun itemClicked(item : PotterCharacter) {
    //create intent
    var intent = Intent()
    intent.action = Intent.ACTION_VIEW
    intent.data = Uri.parse(item.info)

    //start activity
    startActivity(intent)
}
```

The last step in MainActivity is we need to pass this method into the MyListAdapter class as its second parameter. In onCreate() update the line where we define and assign the adapter.

```
val adapter = MyListAdapter(potterList, {item: PotterCharacter ->
itemClicked(item)})
```

This second parameter is a lambda expression. The parameter item is of type Character and the body is the function we just created, and it takes item as a parameter so it knows which item was clicked.

Saving State
Because we load the json data in onCreate() which is called every time the activity starts, this means that we're reading the json file and parsing it every time the device is rotated as well. This is a fairly expensive transaction, so it would be better to only do it once.
We can use the `onSaveInstanceState()` and `onRestoreInstanceState()` methods to save our ArrayList so we don't need to read the json file again.
Since our ArrayList is of a custom type, we can't just use the putString() method, we need to use the Parcelable interface. Parcelable uses the Parcel class which lets us read and write data of various types, including custom types.

To include Parcelable in your project you need to use kotlin extensions which includes the kotlin-parcelize plugin which provides a Parcelable implementation generator.
https://developer.android.com/kotlin/parcelize

In the build.gradle module app file add to the plugins section:
```
id 'kotlin-android-extensions'
```
Then sync.

Now we need to add the @Parcelize annotation to our PotterCharacter class so a Parcelable implementation is automatically generated for that class.

```
@Parcelize
data class PotterCharacter(val name: String, val info: String):
Parcelable {}
```

Now back in MainActivity we're ready to save state. We can use the putParcelableArrayList() and getParcelableArrayList() methods to save our ArrayList of type character.

```
override fun onSaveInstanceState(outState: Bundle) {
    outState.putParcelableArrayList("potterlist", potterList)
    super.onSaveInstanceState(outState)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    potterList =
savedInstanceState.getParcelableArrayList<PotterCharacter>("potterlist
") as ArrayList<PotterCharacter>
}
```

Add some log messages in onCreate(), onRestoreInstanceState(), and getJSON() to make sure that getJSON() is only being called once.
```
Log.i("data", "in on restore")
```

What did you notice?

Since onCreate() is called before onRestoreInstanceState() potterList is always empty and getJSON() is still being called.
Instead, let's test to see if the savedInstanceState that's passed into onCreate() is null, and only load the json data if it's null.

```
if (savedInstanceState == null){
    Log.i("data", "in if")
     val data = JSONData()
     //populate with JSON data
```

```
     potterList = data.getJSON(this)
}
```

Now what's happening?

getJSON() is only being called once but the recyclerView is empty after rotation. That's because the recyclerView and adapter are being set up in onCreate() before onRestoreInstanceState() is being called so potterList is empty.
So let's separate out the recyclerView and adapter set up so we can call that from onRestoreInstanceState() after potterList is populated.
Create a new method that handles the recyclerView and adapter set up.

```
private fun setupRecyclerView(){
    //get the recycler view
    val recyclerView: RecyclerView = findViewById(R.id.recyclerView)

    //divider line between rows
    recyclerView.addItemDecoration(DividerItemDecoration(this,
LinearLayoutManager.VERTICAL))

    //define an adapter
    val adapter = MyListAdapter(potterList, {item: PotterCharacter ->
itemClicked(item)})

    //assign the adapter to the recycle view
    recyclerView.adapter = adapter

    //set a layout manager on the recycler view
    recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)
}
```

And now call this method after we populate potterList which we do at the end of the if statement in onCreate() and onRestoreInstanceState().

```
    setupRecyclerView()
```

Now when you run the app getJSON() should only be called once and the recyclerView should be populated after rotation as well.