Android: Session Four

List views with custom adapters

Here we are going to work and play with Listviews and various related concepts.

Displaying a list in Android involves 4 parts:

- Some data. May sound obvious but we need something to display. How this data is stored is not really relevant here so we are just going to populate a List object with some POKOs (Plain Old Kotlin Object) instances.
- A ListView widget. Also quite obvious, our activity (or fragment) layout must include a widget dedicated to display a list: the ***ListView***.
- A layout resource to set the UI for the rows in our list.
- Now, we need something to tie together all the pieces. This magic component, which will link our data and the listview widget using the row layout is called an *Adapter*.

Now, let's work all this step by step.

First step: Creating the data

Obviously, we need some data to populate the list. In this practical session we are going to make a list of famous characters from movies, or books.

To describe such a character, let us create a MovieCharacter data class. Each character will be described by:

- His (or her) first name
- His last name
- Book or movie title where the character can be seen
- His alignment, telling us if the character is a good guy or a bad guy

```
data class MovieCharacter(
        val firstName: String,
        val lastName: String,
        val movie: String,
        val goodGuy: Boolean)
```

Now, to store our data, we can create a List object (use the ArrayList implementation) of MovieCharacters objects.

Second step: Displaying a list

Maybe the easiest part, to display a list, all we need is to add a ListView widget to our layout.

```
<ListView
    android:id="@+id/activity_main_list_characters"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Et voila !

We shall also add in our layout a special view to show if the list is empty. Any view can be used, but here to keep things simple we will use a TextView.

```
<TextView
    android:id="@+id/activity_main_txt_emptylist"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="This list is empty"
    android:visibility="gone"/>
```

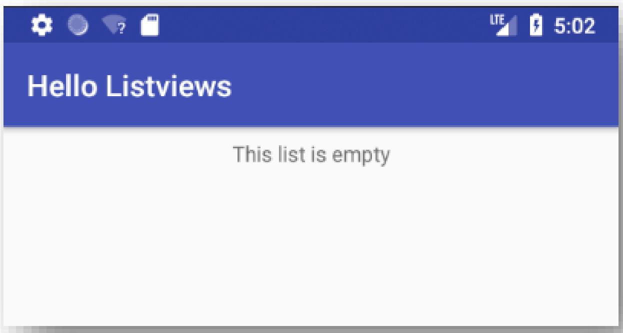
Note the *visibility* attribute? It can take 3 possible values:

- *visible*. This is the default, the view is visible.
- *invisible*. The view is present, occupies some space, but is not visible.
- *gone*. The view is not displayed at all. It does not take any space on the screen, and if it is used as reference point by another view in a RelativeLayout or ConstraintLayout, the constraint will not work.

The basic idea is to switch *visibility* attributes on the list widget and the empty-list view depending on if the list is empty or not. Do not forget to set IDs to your widgets so you can link them in the code.

If you run the application now, you will see the empty-list textview does not show, courtesy of the *visibility="gone"* thing. But we are not going to switch visibilities manually. Instead, the Listview class has a setEmptyView() method we can use.

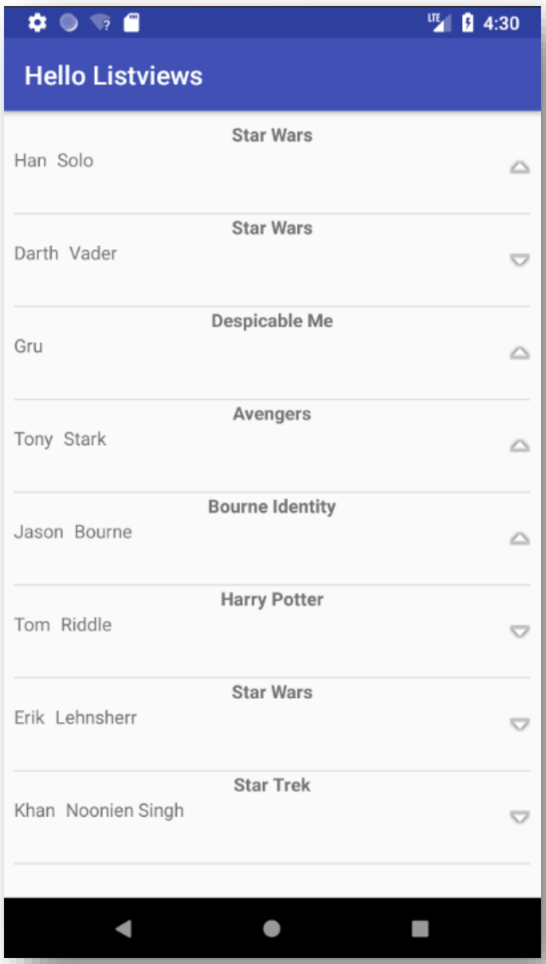
```
activity_main_list_characters.emptyView = activity_main_txt_emptylist
```



```
// In this example, we create data as a hardcoded List
val data : MutableList<MovieCharacter> = arrayListOf()
mdata.add(MovieCharacter("Han", "Solo", "Star Wars", true))
mdata.add(MovieCharacter("Darth", "Vader", "Star Wars", false))
```

Third step: create list row layout

This is the list we want to display:



Hello Listviews

Star Wars
Han Solo

Star Wars
Darth Vader

Despicable Me
Gru

Avengers
Tony Stark

Bourne Identity
Jason Bourne

Harry Potter
Tom Riddle

Star Wars
Erik Lehnsherr

Star Trek
Khan Noonien Singh

As you can see, for each item, we have several pieces of information we want to show. A movie/book/series title, a character firstname and lastname, and an icon indicating if the character is a good guy or a bad guy.

The divider between rows is not part of the row, it is handled directly on the Listview widget with two attributes:

```
<ListView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:divider="#FF0000"
    android:dividerHeight="1dp"/>
```

Knowing that, go on and create a layout resource file to render the row.

N.B: this is the layout to draw one row. And yes, the preview in Android Studio always shows a layout as a screen, but here, try and imagine the preview "screen" is a row.

Fourth step: mix the data, the listview widget, and the row layout

To make this work, the component we are going to use is an adapter. The adapter knows all the pieces of the puzzle, and can bind them together. It will be called by the system whenever a row for our list must be drawn on the screen.

To create our custom adapter, we extend the BaseAdapter class. The BaseAdapter contains several methods but pay special attention to the getView() method, this is where most of the work is done.

```
// our adapter needs to know the data
// we also need a context at some point
class MovieCharacterListAdapter(
        private val context: Context,
        private val data: MutableList<Item>) : BaseAdapter() {

    override fun getCount(): Int {
        // returns the number of items in this adapter
        // usually the size of the underlying List/Array...

        return data.size
    }

    override fun getItem(position: Int): Item {
        // returns the data item at the specified position in the list

        return data[position]
    }

    override fun getItemId(position: Int): Long {
        // return the ID of the specified row (customized or row position)

        return position.toLong()
    }
    // called by the system whenever a row must be drawn on the screen
    // the getView() method receives a position and must return the
    // corresponding row view
    override fun getView(
                    position: Int,
                    convertView: View?,
                    parent: ViewGroup?): View {
        // first let us retrieve the item at the specified position
        val currentItem: Item = getItem(position)

        // now we build a view
        // first step, acquire a LayoutInflater
        val layoutInflater = LayoutInflater.from(context)

        // now use this LayoutInflater to inflate our row layout resource
        // into a View
        val rowView = layoutInflater.inflate(
                R.layout.list_item,
                parent,
                false
        );

        // bind variables to the distinct views inside our row view
        val nameTextView = rowView.findViewById<TextView>(R.id.item_txt_name)

        // finally, time to put the data in here
        nameTextView.text = currentItem.name

        // job’s done, the view is built and contains the data for the
        // requested position, we can return it to the system
        return rowView;
    }

}
```

The up and down arrow drawables used in this application can be found in the Android framework. References to *arrow_up_float* and *arrow_down_float* can be found in the *android.R* class.

```
// Link the adapter and the listview
list_characters.adapter = MovieCharacterListAdapter(activity, data)
```

N.B. *When the data underlying a listview (or recyclerview) is modified, call the* ***notifyDataSetChanged()*** *on its adapter to refresh the view.*

```
data.add(MovieCharacter("Tony", "Stark", "Avengers", true))
(activity_main_list_characters.adapter as BaseAdapter).notifyDataSetChanged()
```

Handle user interaction

Touch (or click) events on a row item can be intercepted by setting an OnItemClickListener on the list.

```
activity_main_list_characters.setOnItemClickListener {
    adapterView, clickedView, position, id ->
    Toast.makeText(activity, "Clicked " + position, Toast.LENGTH_SHORT).show()
}
```

Parameters sent with the event are:

- The listview where the click happened
- The view that was clicked
- The position of the view in the adapter
- The row id of the item that was clicked (here, the same as the position)

Performance patterns

First step: the convertview

The getView() is going to be called by the system for every row that has to be displayed. That is to say: a lot. And even more during scrolling. This is why no heavy operation must be performed in the getView().

The major flaw in our custom BaseAdapter getView() is that each time, we create (and inflate !) a new view. Inflating a layout into a view IS a heavy operation.
Moreover, when the list scrolls, not all rows are displayed at the same time. For example, if the list contains 200 characters, you can only see 9 or 10 at once.

This is why the convert view system was added. The basic idea is that whenever a row exits the screen, instead of destroying its view, the Android system will keep it in a kind of recycling trashcan. And when a new row enters the screen, before calling getView(), the system will check if any such recycled views are available. In which case the view will be passed to the getView and we do not need to create a new one!

```
override fun getView(
        position: Int,
        convertView: View?,
        parent: ViewGroup?): View {
    val currentItem: MovieCharacter = getItem(position)
    val rowView: View

    // before creating a new view, check if the input convertView is null
    if (convertView == null) {
        rowView = LayoutInflater.from(context).inflate(
                R.layout.list_item_moviecharacter,
                parent,
                false
        )
    } else {
        rowView = convertView
    }

    // back to previous situation, we have a row view to return
    // do stuff like…
    val nameView = rowView.findViewById<TextView>(R.id.item_name)

    return rowView;
}
```

N.B. the convertview you get is not fresh and empty. It is a view you built and used before, so it already contains data when you get it. Before you use it, make sure you set, or reset, all its constituting elements.

Second Step : the holder pattern

Using the convertview is better, but another tiny little problem remains. Every time a row is displayed, we search the row view and create new references to its composing views. In our, albeit simple, example, 4 calls to *findViewById()*, and as many references to new objects are created each time.

The idea behind the ViewHolder pattern is to find a way to store those references inside the view object itself! Thus, when getting a row convert view, we can also retrieve the views inside it.

To store our references, we are going to use the *tag* property of our rowView object. The type is Any so basically, we can store anything in there!

All we need to do is create an object that will hold all our references for us: the viewHolder !

Declare the holder as class in your adapter:

```
class ViewHolder(
     val firstnameTextView: TextView,
     val lastnameTextView: TextView)
```

Now you can use this class in the getView() method:

```
override fun getView(
        position: Int,
        convertView: View?,
        parent: ViewGroup?): View {
    val currentItem: MovieCharacter = getItem(position)
    val rowView: View
    val viewHolder : ViewHolder

    if (convertView == null) {
        rowView = LayoutInflater.from(context).inflate(
                R.layout.list_item_moviecharacter,
                parent,
                false
        )
        // assign the new ViewHolder object
        viewHolder = ViewHolder(
            rowView.findViewById(R.id.item_txt_firstname),
            rowView.findViewById(R.id.item_txt_lastname))
        // stash it inside the view
        rowView.tag = viewHolder
    } else {
        rowView = convertView
        // extract our ViewHolder object from the convertView
        viewHolder = rowView.tag as ViewHolder
    }

    // no need to find views by id anymore, references are stored inside
    // the viewHolder so we can do stuff like...
    viewHolder.firstnameTextView.text = currentItem.firstname

    return rowView;
}
```

RecyclerView is the new ListView

Since pattern performances are indeed very important, a new list widget has been added to the framework, the RecyclerView.

Create a new HelloRecycler project.

Unlike the ListView, RecyclerView is not part of the Android SDK and is rather distributed as a library. To be able to use it you must first add it to your project.
From your HelloRecycler project tree view, open the *build.gradle* file (the one for the **app** module).
Add a dependency to the recyclerview library:

```
dependencies {
     …
    // Add recyclerview to module
    implementation 'com.android.support:recyclerview-v7:26.1.0'
}
```

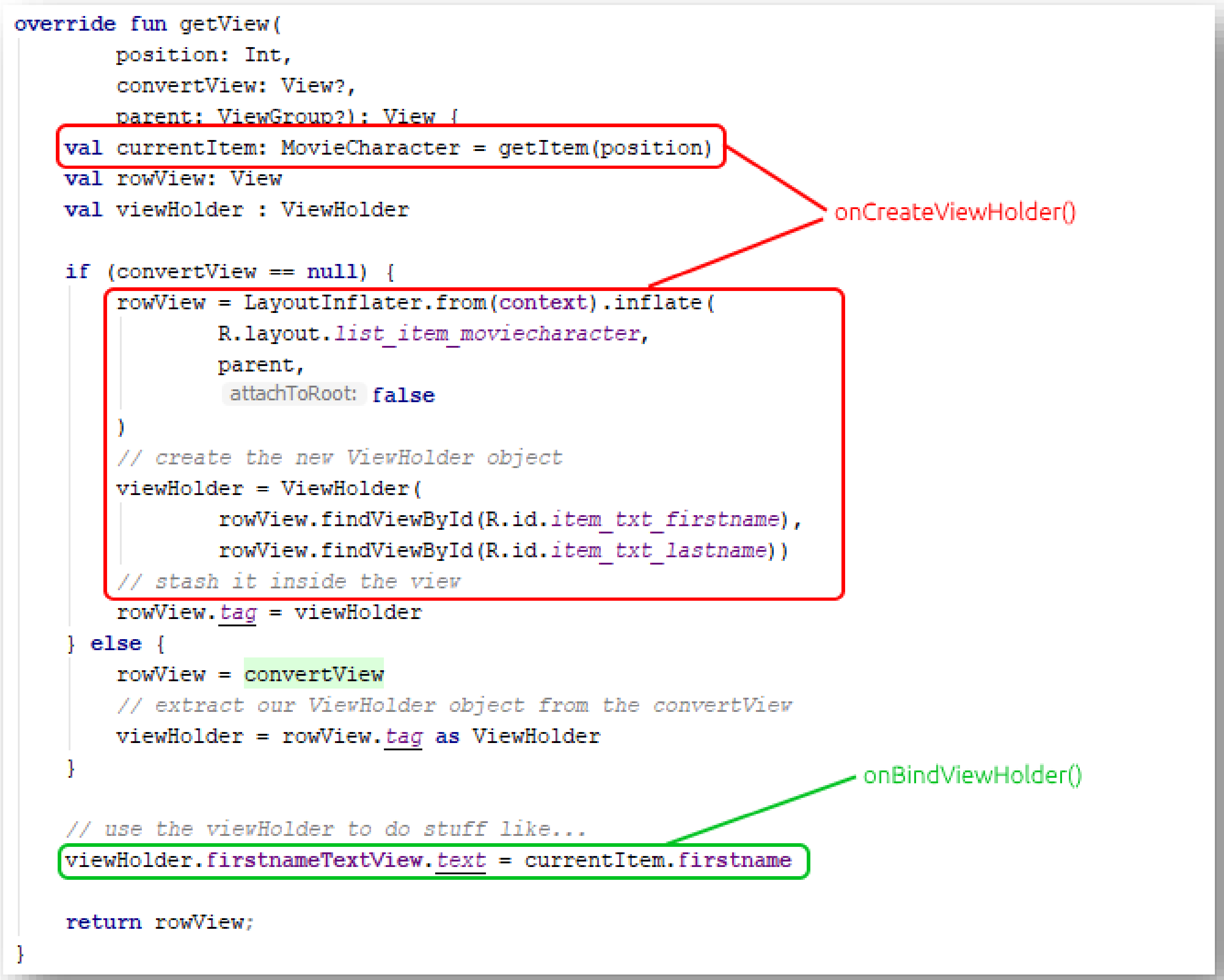
N.B. when you modify a *build.gradle* file, do not forget to sync it in order to take your modifications into account.

Now, you can add a Recycler view in your layout.

```
<android.support.v7.widget.RecyclerView
    android:layout_width="0dp"
    android:layout_height="0dp"/>
```

Just like the listview, the recycler view requires an adapter to feed it with data.
The structure for a recycler adapter is quite different from the BaseAdapter. Instead of revolving around the getView() method, the new RecyclerAdapter works solely with the ViewHolder idea.

The getView() is now split in two: stuff that is done only once per displayed row goes inside the onCreateViewHolder() method, and stuff that is done everytime a row is displayed goes in the onBindViewHolder() method. What is left is now done by the adapter (nice, hey?)



```
override fun getView(
        position: Int,
        convertView: View?,
        parent: ViewGroup?): View {
    val currentItem: MovieCharacter = getItem(position)
    val rowView: View
    val viewHolder : ViewHolder

    if (convertView == null) {
        rowView = LayoutInflater.from(context).inflate(
                R.layout.list_item_moviecharacter,
                parent,
                attachToRoot: false
        )
        // create the new ViewHolder object
        viewHolder = ViewHolder(
                rowView.findViewById(R.id.item_txt_firstname),
                rowView.findViewById(R.id.item_txt_lastname))
        // stash it inside the view
        rowView.tag = viewHolder
    } else {
        rowView = convertView
        // extract our ViewHolder object from the convertView
        viewHolder = rowView.tag as ViewHolder
    }

    // use the viewHolder to do stuff like...
    viewHolder.firstnameTextView.text = currentItem.firstname

    return rowView;
}
```

```
class CustomRecyclerAdapter(
              val context: Context,
              val data: MutableList<Item>) :
        RecyclerView.Adapter<CustomRecyclerAdapter.ViewHolder>() {

    // the new RecyclerAdapter enforces the use of
    // the ViewHolder class performance pattern
    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val nameTextView: TextView = itemView.findViewById(R.id.name)
        val logoImageView: ImageView = itemView.findViewById(R.id.logo)
    }

    override fun getItemCount(): Int {
        return data.size
    }

    // called when a new viewholder is required to display a row
    override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int)
                    : ViewHolder {
        // create the row from a layout inflater
        val rowView = LayoutInflater
                .from(context)
                .inflate(R.layout.list_item, parent, false)

        // create a ViewHolder using this rowview
        val viewHolder = ViewHolder(rowView)

        // return this ViewHolder. The system will make sure view holders
        // are used and recycled
        return viewHolder
    }

    // called when a row is about to be displayed
    override fun onBindViewHolder(holder: ViewHolder?, position: Int) {
        // retrieve the item at the specified position
        val currentItem = data[position]

        // put the data
        holder!!.nameTextView.text = currentItem.name
        holder.alignmentImageView.setImageResource(R.drawable.logo)
    }

}
```

Now that the adapter class is available, the recycler view can be fully set:

```
// display performance optimization when list widget size does not change
activity_main_list_characters.setHasFixedSize(true)

// here we specify this is a standard vertical list
activity_main_list_characters.layoutManager = LinearLayoutManager(
        activity,
        LinearLayoutManager.VERTICAL,
        false)

// as with the listview, attach an adapter and provide some data
activity_main_list_characters.adapter = CustomRecyclerAdapter(activity, data)
```

Handle user interaction

The RecyclerView does not provide a setOnItemClick listener like the ListView, so we must add it ourselves.

The simplest way of doing this is to add a View.OnClickListener property to our fragment, pass it along to the adapter, and bind it to the item view when it is created.

In the fragment:

```
val myItemClickListener = View.OnClickListener {
    // we retrieve the row position from its tag
    val position = it.tag as Int
    val clickedItem = data[position]

    // do stuff
    Toast.makeText(
             activity,
             "Clicked " + clickedItem.name,
             Toast.LENGTH_SHORT)
          .show()
}

// as with the listview, attach an adapter and provide some data
val recyclerAdapter = CustomRecyclerAdapter(activity, data, myItemClickListener)
activity_main_list_characters.adapter = recyclerAdapter
```

And modify the Recycler adapter accordingly.

```
class CustomRecyclerAdapter(
        val context: Context,
        val data: MutableList<MovieCharacter>,
        private val onItemClickListener: View.OnClickListener) :
        RecyclerView.Adapter<CustomRecyclerAdapter.ViewHolder>() {

    // the new RecyclerAdapter enforces the use of the ViewHolder performance pattern
    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val nameTextView: TextView = itemView.findViewById(R.id.name)
    }

    override fun getItemCount(): Int {
        return data.size
    }

    // called when a new viewholder is required to display a row
    override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int): ViewHolder {
        // create the row from a layout inflater
        val rowView = LayoutInflater
                .from(context)
                .inflate(R.layout.list_item, parent, false)

        // attach the onclicklistener
        rowView.setOnClickListener(onItemClickListener)

        // create a ViewHolder using this rowview
        val viewHolder = ViewHolder(rowView)

        // return this ViewHolder. The system will make sure view holders
        // are used and recycled
        return viewHolder
    }

    // called when a row is about to be displayed
    override fun onBindViewHolder(holder: ViewHolder?, position: Int) {
        // retrieve the item at the specified position
        val currentItem = data[position]

        // put the data
        holder!!.nameTextView.text = currentItem

        // store row position inside view tag
        holder.itemView.tag = position
    }

}
```