Android Development with Kotlin

Recycler View in Android

What is a Recycler View?

RecyclerView is a ViewGroup added to the android studio as a successor of the GridView and ListView. It is an improvement on both of them and can be found in the latest v-7 support packages. It has been created to make possible construction of any lists with XML layouts as an item which can be customized vastly while improving on the efficiency of ListViews and GridViews. This improvement is achieved by recycling the views which are out of the visibility of the user. For example, if a user scrolled down to a position where the items 4 and 5 are visible; items 1, 2 and 3 would be cleared from the memory to reduce memory consumption.
The List View can be customized to follow the same behaviour as a Recycler View by using the View Holder Design pattern. The only difference is that, the View Holder pattern is optional in a List View where as it becomes mandatory in a Recycler View.

Adding a Recycler View

RecyclerView can be used within an Android project once its dependencies have been added in the build.gradle file. This can be done using the following statement.

```
dependencies {
      implementation 'com.android.support:recyclerview-v7:28.0.0'
}
```

After the addition of the dependencies, the view can be added to an Activity using the following code in its XML layout:

```
<android.support.v7.widget.RecyclerView
      android:id="@+id/my_recycler_view"
      tools:listitem = "@layout/list_item"
      android:scrollbars="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
```

The list item attribute allows the developer to visualize the list layout during the design time itself.

A Recycler view has two parameters that are important for its population, a Layout Manager that arranges the views in a specified format, and the adapter that handles the population of those views with data.

Layout Manager

A LayoutManager is responsible for measuring and positioning item views within a RecyclerView as well as determining the policy for when to recycle item views that are no longer visible to the user. By changing the LayoutManager a RecyclerView can be used to implement a standard vertically scrolling list, a uniform grid, staggered grids, horizontally scrolling collections and more. RecyclerView uses pre built LayoutMangers and Implements different behaviours such as :

1. Vertical/Horizontal List(Using LinearLayoutManger)
2. Uniform Grid List (Using GridLayoutManager)
3. Staggered Grid List (Using StaggeredGridLayoutManager)

The following code segment depicts how to initialize a LinearLayout Manager for a RecyclerView in Kotlin:

```
var viewManager = LinearLayoutManager(this)
```

The LinearLayoutManager takes in one parameter that is the context of the application, here the context of the current Activity is passed as “this”.

Creating Adapters

To feed all the data to the list, the created adapter must extend the RecyclerView.Adapter class. This object creates views for items, and replaces the content of some of the views with new data items when the original item is no longer visible.
The following code example shows a simple implementation for a data set that consists of an array of strings displayed using a TextView.

```
class MyAdapter(private val myDataset: Array<String>) :
        RecyclerView.Adapter<MyAdapter.MyViewHolder>() {
        class MyViewHolder(val textView: TextView) :
                        RecyclerView.ViewHolder(textView)
        // Create new views (invoked by the layout manager)
        override fun onCreateViewHolder(parent: ViewGroup,
                        viewType: Int): MyAdapter.MyViewHolder {
                // create a new view
                val textView = LayoutInflater.from(parent.context)
         .inflate(R.layout.my_text_view, parent, false) as TextView

                return MyViewHolder(textView)
        }

        // Replace the contents of a view
        // (invoked by the layout manager)
        override fun onBindViewHolder(holder: MyViewHolder,
                        position: Int) {
                //get element from your dataset at this position
                //replace the contents of the view with that element
                holder.textView.text = myDataset[position]
        }

        // Return the size of your dataset
        // (invoked by the layout manager)
        override fun getItemCount() = myDataset.size
}
```

The custom adapter class extends a RecyclerView.Adapter which is a generic class which requires ViewModel as its input. The custom viewmodel can be created as per the requirements of the application ut the basic structure of the adapter remains the same.

Whenever the list changes, the adapter needs to be notified using the "notifyDatasetChanged()" function so as to reflect the changes in the list as the adapter isn't smart enough to reflect those changes on its own.
If you want to ensure that the list modifications are reflected automatically, the custom adapter should extend a ListAdapter instead. This adapter is capable of reflecting changes on its own with minimum modifications. The list adapter can be defined as follows:

```
class CustomAdapter: ListAdapter<User, CustomAdapter.ItemViewholder>
                              (DiffCallback())  {

      override fun onCreateViewHolder(parent: ViewGroup,
                              viewType: Int): ItemViewholder {
                  //can be same as before
      }

      override fun onBindViewHolder
            (holder: CustomAdapter.ItemViewholder, position: Int) {
                  //can be same as before
      }

      class ItemViewholder(itemView: View) :
            RecyclerView.ViewHolder(itemView) {
                  //can be same as before
      }
}
class DiffCallback : DiffUtil.ItemCallback<User>() {
      override fun areItemsTheSame(oldItem: User?,
                        newItem: User?): Boolean {
            return oldItem?.id == newItem?.id
      }

      override fun areContentsTheSame(oldItem: User?,
                        newItem: User?): Boolean {
            return oldItem == newItem
      }
}
```

The only differences are that the ListAdapter takes in two classes for its generic types, the Data Class as well as a ViewHolder and has an object of a callback that implements two functions to check if two data instances are the same, or if two data instances store the same data.
The objects of RecyclerViews can be created using the following code segment:

```
viewManager = LinearLayoutManager(this)
viewAdapter = MyAdapter(myDataset)

recyclerView = findViewById<RecyclerView>(R.id.my_recycler_view)
        .apply {
                setHasFixedSize(true)
                layoutManager = viewManager
                adapter = viewAdapter
        }
```

To study more about RecyclerViews you can visit
https://developer.android.com/guide/topics/ui/layout/recyclerview