Android Development with Kotlin

List View and SharedPreference in Android

List View

A list view displays a vertically-scrollable collection of views, where each view is positioned immediately below the previous view in the list. To display a list in and android application, a list view is added to the xml file using the following code:

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

A list view is an adapter view that does not know the details, such as type and contents, of the views it contains. Instead list view requests views on demand from a ListAdapter as needed, such as to display new views as the user scrolls up or down.
In order to display items in the list, call setAdapter(android.widget.ListAdapter) to associate an adapter with the list. This is done by adding the following code in the onCreate() function.

```
val listView = findViewById(R.id. list_view)
val listItems = arrayOf("New Delhi", "New York", "Paris", "Rome",
                "Amsterdam")
val adapter = ArrayAdapter(this,
                android.R.simple_list_item_1, listItems)
listView.adapter = adapter
listView.setOnItemClickListener {adapterView, View, position, id ->
    val city: TextView = view as TextView
    Toast.makeText(this, city.text, Toast.LENGTH_SHORT).show()
}
```

The arguments for the ArrayAdapter in the above-mentioned code are the context of the Activity, the layout id for the list view and the array of content. A simple List Adapter uses text view as the default list view but a custom view and adapter can be used as well, this can be achieved by creating a custom adapter class which extends the BaseAdapter. **When setting the onClickListener using the entire adapter, the click event is registered for the entire view, this too can be altered with the use of a custom adapter wherein the list item view can be a complex layout in itself with a number of elements with different event listeners for each and every one of them.**
The custom adapters can be defined as follows:

```
class CustomAdapter(private val context: Context,
            private    val    dataList:    ArrayList<String>)    :
BaseAdapter() {

    private val inflater: LayoutInflater =
            this.context.getSystemService(Context.LAYOUT_INFLATER_SER
            VICE) as LayoutInflater
    override fun getCount(): Int
      { return dataList.size }
    override fun getItem(position: Int): String
      { return dataList[position] }
    override fun getItemId(position: Int): Long
      { return position.toLong() }

    override fun getView(position: Int, convertView: View?,
                         parent: ViewGroup): View {
        var dataitem = dataList[position]
```

```
            if(convertView == null)
                    convertView = inflater.inflate(R.layout.list_row,
                                    parent, false)

               convertView.findViewById<TextView>(R.id.row_name).text =
                                                        dataitem

            convertView.tag = position
            return convertView
        }
    }
```

The custom adapter has to override the getCount(), getItem(), getItemId(), and the getView() functions. The getView() function inflates a view every time the convertView is null or the screen doesnot have enough views for the list items to be displayed at the moment.
"convertView" is null if a reusable view doesn't exist. It takes the value of the reusable view otherwise. This reduces time lag as the views don't have to be rendered for all list item. ***Only n+1 views are generated where n is the number of views visible on the screen at any given time.*** When the user scrolls, the same convertView is reused, but the "findViewbyId()" functions are reinvoked, which can lead to a time lag when scrolling. This can be rectified using the ViewHolder design pattern. Each convertView can hold a tag (in this case the position of the item in the list).
The ViewHolder design pattern enables you to access each list item view without the need for the look up, saving valuable processor cycles. Specifically, it avoids frequent call of findViewById() during ListView scrolling, and that will make it smooth. The following code segment explains how ViewHolders can be used with List Views in Kotlin:

1. Create a ViewHolder class inside the adapter to hold the Views for each list item.

```
private class ViewHolder(view: View?) {
        val itemName = view?.findViewById(R.id.row_name)
                        as TextView
}
```

2. Bind a ViewHolder to a convertView in the getView() function, and reuse the viewholders to avoid the frequent findViewById calls.

```
    override fun getView(position: Int, convertView: View?,
    parent: ViewGroup): View {
        var dataitem = dataList[position]

            if(convertView == null){
                  convertView = inflater.inflate(R.layout.list_row,
                                parent, false)
              val viewHolder = ViewHolder(convertView)
              convertView.tag = viewHolder
            }

            val viewHolder = convertView.tag as ViewHolder

               viewHolder.itemName = dataitem
            return convertView
        }
```

Shared Preferences

Android provides many ways of storing data of an application. One of this way is called Shared Preferences. Shared Preferences allows one to save and retrieve data in the form of key-value pairs. In order to use shared preferences, we have to call a method getSharedPreferences() that returns a SharedPreference instance pointing to the file that contains the values of preferences. The first parameter is the key or the name of the Shared Preferences file and the second parameter is the MODE. The following modes are available:

- **MODE_APPEND** This will append the new preferences with the already existing preferences
- **MODE_ENABLE_WRITE_AHEAD_LOGGING** Database open flag. When it is set, it would enable write ahead logging by default
- **MODE_MULTI_PROCESS** This method will check for modification of preferences even if the sharedpreference instance has already been loaded
- **MODE_PRIVATE** By setting this mode, the file can only be accessed using calling application
- **MODE_WORLD_READABLE** This mode allows other application to read the preferences
- **MODE_WORLD_WRITEABLE** This mode allows other application to write the preferences

For any particular set of preferences, there is a single instance of this class that all clients share. Modifications to the preferences must go through an Editor object to ensure the preference values remain in a consistent state and control when they are committed to storage. Objects that are returned from the various get methods must be treated as immutable by the application. The following code segment illustrates how the data can be saved in a Shared Preference in Kotlin:

```
 val sharedPref: SharedPreferences =
       getSharedPreferences("data",SharedPrefrences.MODE_PRIVATE)
val editor = sharedPref.edit()          //creates the editor
editor.putBoolean("boolean", true)      //adds a boolean
editor.apply()
```

An editor needs to be created to modify the contents in a Shared Preference, but the data can be accessed without one, using the following code segment:

```
val b = sharedPref.getBoolean("Boolean", false)
```

Here, the first parameter is the key for the Boolean and the second parameter is a default value to be used if the Boolean with the given key is absent within the Shared Preference. The shared preference can be used to store any primitive datatype as well as objects of a serializable class.
Shared Preferences can be used to store small amounts of data such as the login information of the user, the token id of the login session, high scores for a game etc. In other words ***SharedPreferences are a perfect way to preserve history in an application.***