Layouts and views:

There are number of Layouts provided by Android:

- LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
- RelativeLayout is a view group that displays child views in relative positions.
- TableLayout is a view that groups views into rows and columns.
- AbsoluteLayout enables you to specify the exact location of its children.
- FrameLayout is a placeholder on screen that you can use to display a single view.
- ListView is a view group that displays a list of scrollable items.
- GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

Additionally, there are Layouts delivered by support library (jetpack):

- RecyclerView is a view group that displays a list of scrollable items and has much better performance than ListView. It is described in detail in this instruction.
-

Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes:

- android:id – this is the ID which uniquely identifies the view.
- android:layout_width – this is the width of the layout.
- android:layout_height – this is the height of the layout
- android:layout_marginTop, android:layout_marginBottom, android:layout_marginLeft, android:layout_marginRight – they are the extra spaces on the specific side of the layout.
- android:layout_gravity – this specifies how child Views are positioned.
- android:layout_weight – this specifies how much of the space in the layout (linear) should be allocated to the View.
- android:paddingLeft, android:paddingRight, android:paddingTop, android:paddingBottom – they is the bottom padding filled for the layout.

Sizes

Android devices come in different screen sizes (handsets, tablets, TVs, and so on), but their screens also have different pixel sizes. That why we need to use density-independent pixels (dp) for sizes and scale-independent pixels (sp) for font size.
The conversion of dp units to screen pixels is simple:
px = dp * (dpi / 160)
Sp additionaly depends on font size settings (general device settings).
Remember to always use dp or sp for size/font size values.
There is recommendation to also extract all these values to dimens.xml in res directory.

RecyclerView

RecyclerView is a UI component which allows us to create scrolling list. It was introduced with the Android Lollipop and is used instead ListView.

RecyclerView is a subclass of ViewGroup and is a more resource-efficient way to display scrollable lists. Instead of creating a View for each item that may or may not be visible on the screen, RecyclerView creates a limited number of list items and reuses them for visible content.

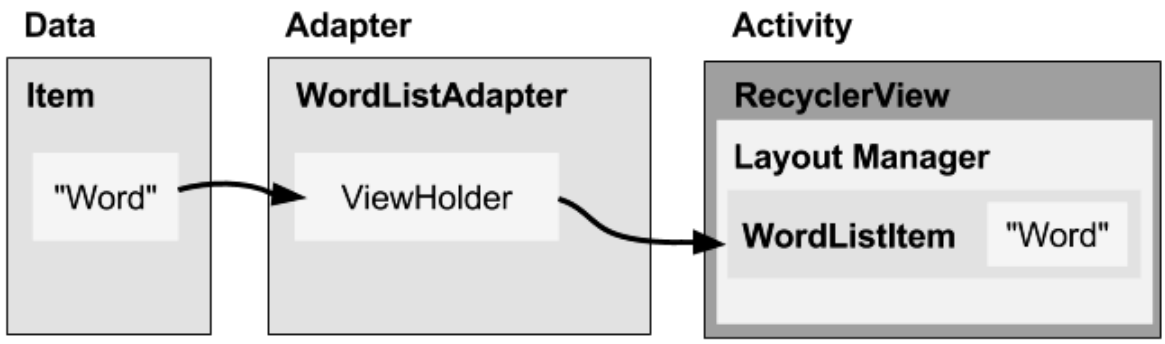
Compared to old ListView

RecyclerView differs from its predecessor ListView primarily because of the following features:

- Required ViewHolder in Adapters - ListView adapters do not require the use of the ViewHolder pattern to improve performance. In contrast, implementing an adapter for RecyclerView requires the use of the ViewHolder pattern for which it uses RecyclerView.Viewholder.
- Customizable Item Layouts - ListView can only layout items in a vertical linear arrangement and this cannot be customized. In contrast, the RecyclerView has a RecyclerView.LayoutManager that allows any item layouts including horizontal lists or staggered grids.
- Easy Item Animations - ListView contains no special provisions through which one can animate the addition or deletion of items. In contrast, the RecyclerView has the RecyclerView.ItemAnimator class for handling item animations.
- Manual Data Source - ListView had adapters for different sources such as ArrayAdapter and CursorAdapter for arrays and database results respectively. In contrast, the RecyclerView.Adapter requires a custom implementation to supply the data to the adapter.
- Manual Item Decoration - ListView has the android:divider property for easy dividers between items in the list. In contrast, RecyclerView requires the use of a RecyclerView.ItemDecoration object to setup much more manual divider decorations.
- Manual Click Detection - ListView has a AdapterView.OnItemClickListener interface for binding to the click events for individual items in the list. In contrast, RecyclerView only has support for RecyclerView.OnItemTouchListener which manages individual touch events but has no built-in click handling.

To implement a RecyclerView we have to create the following:

- a RecyclerView which we should add to our screen layout,
- a layout for each row in the list,
- an adapter which holds the data and bind them to the list.



First you need to add a dependency to the build.gradle. Remember to update the library version to the most recent one. (https://developer.android.com/topic/libraries/support-library/packages.html)

```
class MyActivity : Activity() {
    private lateinit var recyclerView: RecyclerView
    private lateinit var viewAdapter: RecyclerView.Adapter<*>
    private lateinit var viewManager:
                        RecyclerView.LayoutManager

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

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

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

On the created the RecyclerView instance we have to set a LayoutManager. Following the documentation:

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.

So basically, it means that the LayoutManager is layouting the list in a chosen way. In this example we've used LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically). But it can be changed in a very simple way just by calling different one, like GridLayoutManager, StaggeredGridLayoutManager or event WearableLinearLayoutManager.

We have to create a class which extends RecyclerView.Adapter which as a parameters takes a class which extends RecyclerView.ViewHolder. Another thing is to override some needed methods.
What are these methods doing?

- getItemCount() returns the total number of the list size. The list values are passed by the constructor.
- onCreateViewHolder() creates a new ViewHolder object whenever the RecyclerView needs a new one. This is the moment when the row layout is inflated, passed to the ViewHolder object and each child view can be found and stored.

- onBindViewHolder() takes the ViewHolder object and sets the proper list data for the particular row on the views inside.

Basic usage of RecyclerView:

```
class MainActivity: AppCompatActivity() {

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

        val recyclerView: RecyclerView = findViewById(R.id.recyclerview)
        recyclerView.layoutManager = LinearLayoutManager( context: this)
        recyclerView.adapter = Adapter(generateFakeValues())
    }

    private fun generateFakeValues(): List<String> {
        val values = mutableListOf<String>()
        for(i in 0..100) {
            values.add("$i element")
        }
        return values
    }

    class Adapter(private val values: List<String>): RecyclerView.Adapter<Adapter.ViewHolder>() {

        override fun getItemCount() = values.size

        override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int): ViewHolder {
            val itemView = LayoutInflater.from(parent?.context).inflate(R.layout.list_item_view, parent,  attachToRoot: false)
            return ViewHolder(itemView)
        }

        override fun onBindViewHolder(holder: ViewHolder?, position: Int) {
            holder?.textView?.text = values[position]
        }

        class ViewHolder(itemView: View?) : RecyclerView.ViewHolder(itemView) {
            var textView: TextView? = null
            init {
                textView = itemView?.findViewById(R.id.text_list_item)
            }
        }
    }
}
```

Additionally: ItemAnimator
RecyclerView.ItemAnimator will animate ViewGroup modifications such as add/delete/select that are notified to the adapter. DefaultItemAnimator can be used for basic default animations and works quite well. See the section of this guide for more information.

Exercise:

Write a tasks manager app in which:

- (1p.) User can add at least 4 types of tasks (for example: todo, email, phone, meeting etc.)
- (2p.) Each task has:
    - Title,
    - Description,
    - Due date,
    - Status (done/ not done)
    - Icon (depends on type).
- (1p.) Application should have action bar with “+” action to add new task.
- (2p.) Main app screen include list of all tasks. Each list item should have icon, title, due date and status.
- (1p.) After click on the item user see new screen with all data about selected task.

- (1p.) Swipe left on item mark it as done.
- (1p.) Swipe right remove the specific item.