Displaying Large Amount of Data

We love scrolling

- Real-world apps often have lots of data in a scrolling list.

src: https://www.eyerys.com/

UIs for Large Amounts of Data

- First, managing this is not like getting/setting EditText.
- But why not? In another reality, things could be simpler:
  - A “BigListView” UI element (not a real thing) could display a scrolling list.
  - It could have assorted get, set, add and remove methods.
  - It would display strings (or Objects, by calling toString()).
- Nice and easy, except. . .
  - UI lists are not just rows of strings.
    - Each row may contain several text fields, buttons, etc.
  - You don’t need more UI objects than you can actually display.
    - You may have a million data elements.
    - But, at one time, maybe only 12 of them fit on the screen.
    - Having a million UI list rows (and all the UI elements inside them) just wastes memory

Create dynamic lists with RecyclerView

RecyclerView makes it easy to efficiently display large sets of data.

You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements **when they're needed**.

When an item scrolls off the screen, RecyclerView doesn't destroy its view. Instead, RecyclerView reuses the view for new items that have scrolled onscreen.

RecyclerView improves performance and your app's responsiveness, and it reduces power consumption.

scr: https://developer.android.com/develop/ui/views/layout/recyclerview

Key classes

RecyclerView is the ViewGroup that contains the views corresponding to your data. It's a view itself, so you add RecyclerView to your layout the way you would add any other UI element.

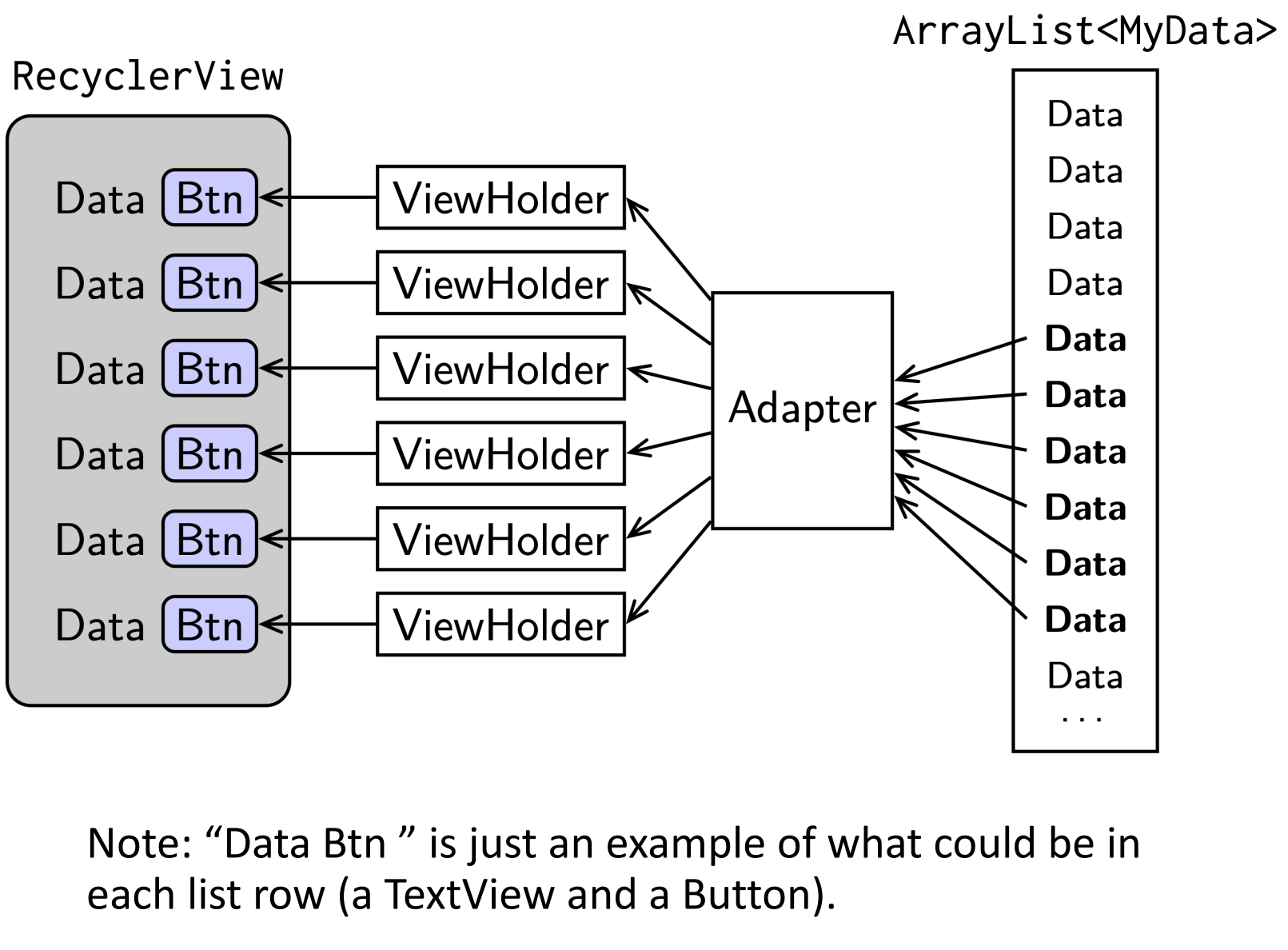
Each individual element in the list is defined by a **view holder** object. When the view holder is created, it doesn't have any data associated with it. After the view holder is created, the RecyclerView binds it to its data. You define the view holder by extending **RecyclerView.ViewHolder**.

The RecyclerView requests views, and binds the views to their data, by calling methods in the **adapter**. You define the adapter by extending **RecyclerView.Adapter**.

The layout manager arranges the individual elements in your list. You can use one of the layout managers provided by the RecyclerView library, or you can define your own. Layout managers are all based on the library's **LayoutManager** abstract class.

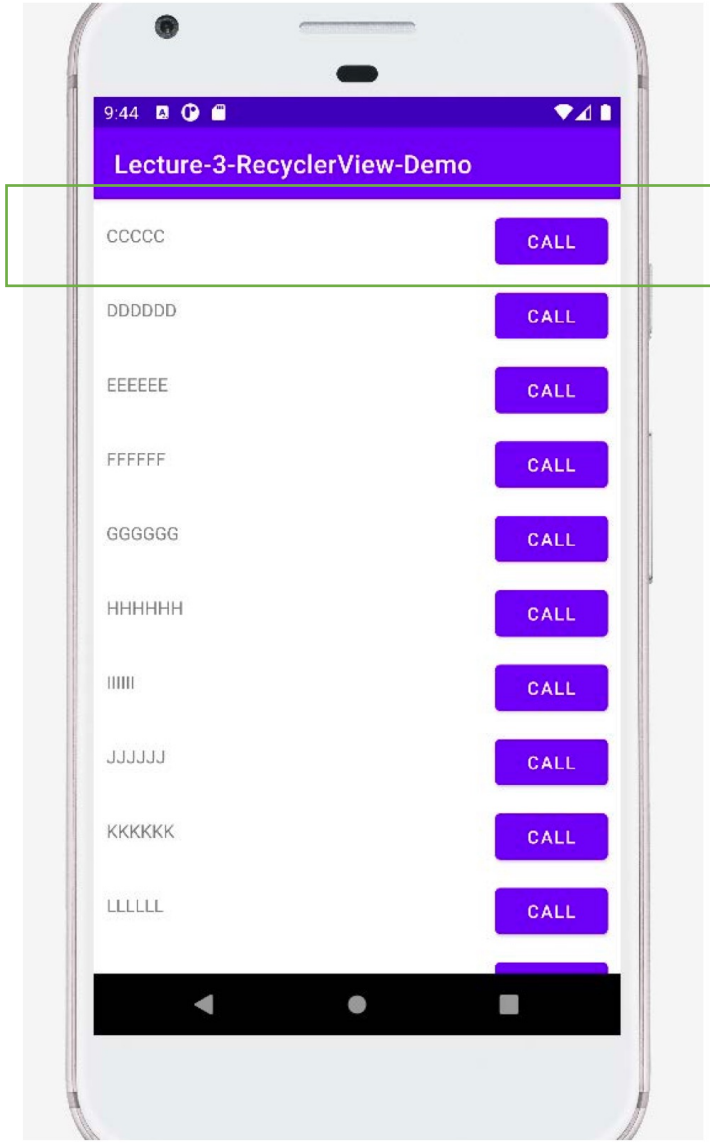
scr: https://developer.android.com/develop/ui/views/layout/recyclerview

Adapters and ViewHolders: How They Interact



Note: “Data Btn ” is just an example of what could be in each list row (a TextView and a Button).

Best way to learn is to do step by step demo



A cell with list items

Step 1: Add RecyclerView to Dependencies

```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.5.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
}
```

Open the build.gradle file (Module-level) and add the RecyclerView dependency to the dependencies block.

Don’t forget to Sync the gradle!!

Add a MyData.Java in the project

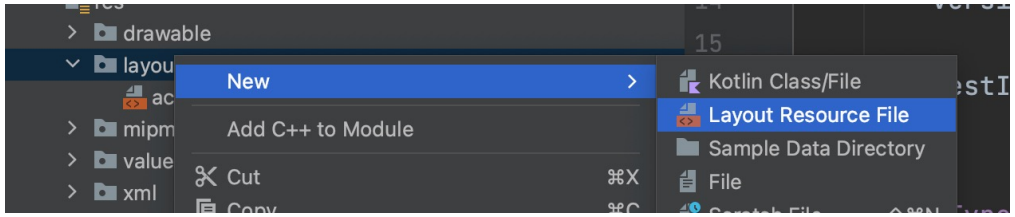
Step 2: Defining the data structure for list items in a cell.

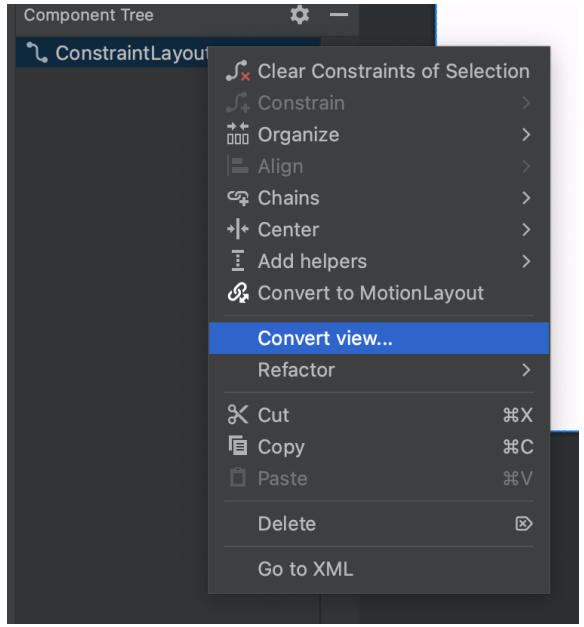
```
public class MyData {

    private String name;
    private String phoneNumber;
    public MyData(String name, String phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

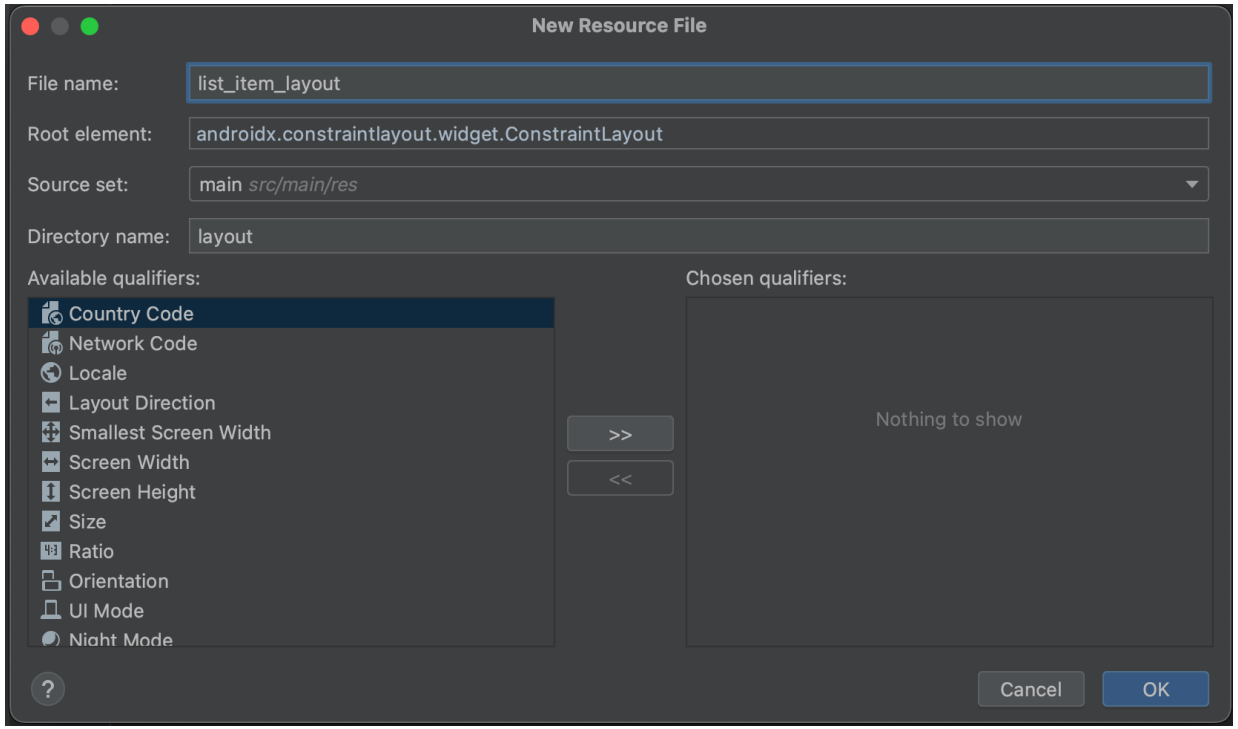
Step 3: Create Layout for list items in a cell

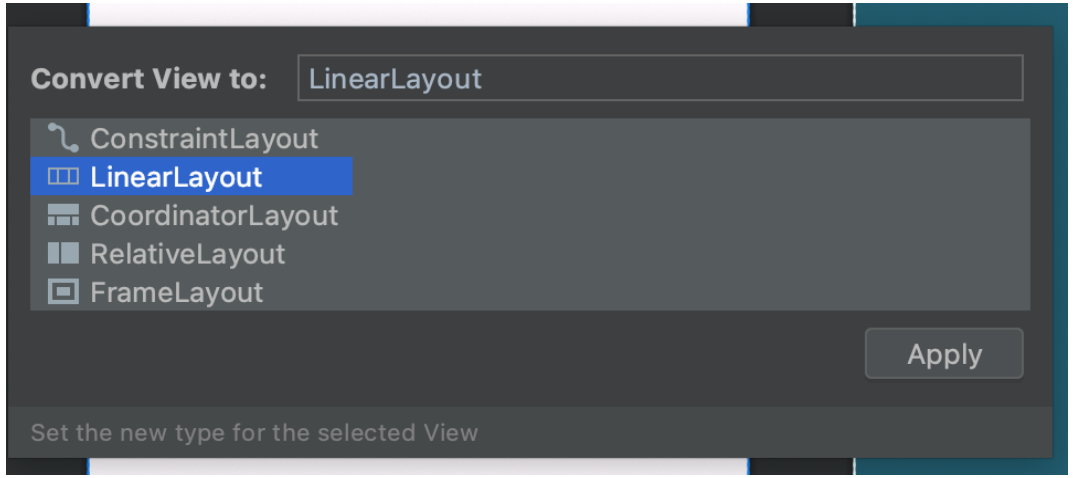
- Create an XML layout file (list_item_layout.xml) to define the layout for individual items in the RecyclerView. Customize this layout as per your requirements.

We are using LinearLayout

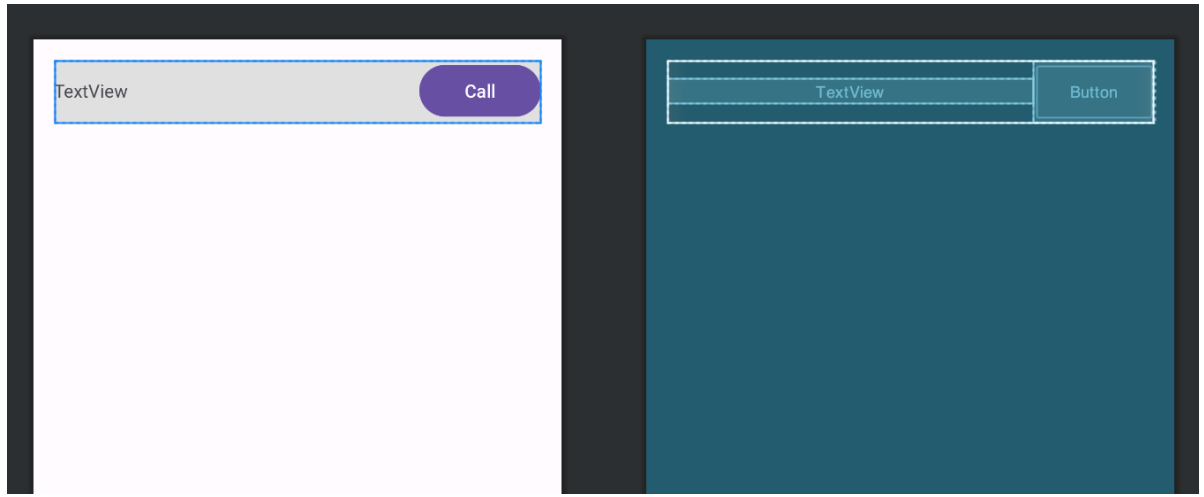








list_item_layout.xml



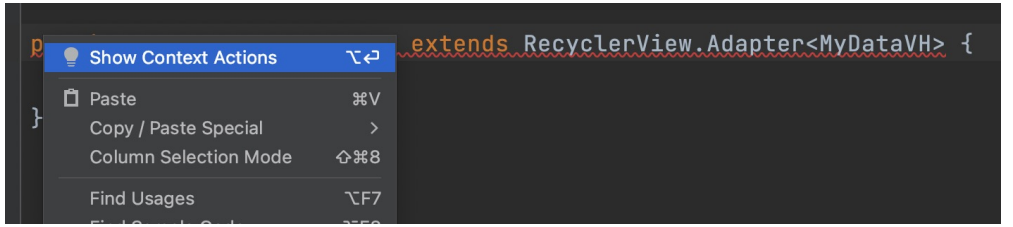

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.an
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:background="#E0E0E0">

    <TextView
        android:id="@+id/textView"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="3"
        android:text="TextView" />

    <Button
        android:id="@+id/button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Call" />
</LinearLayout>
```

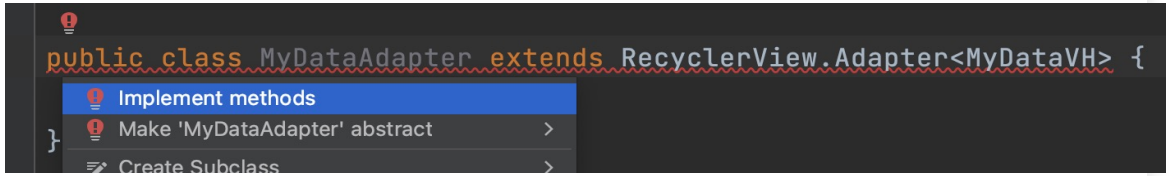
Step 4: Create ViewHolder – MyDataVH.java

- Create a Java class to hold references to the views in the list item layout. This class will extend RecyclerView.ViewHolder.

```java
package com.example.recycleviewdemo;

import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;

public class MyDataVH extends RecyclerView.ViewHolder{
    1 usage
    public TextView nameTextBox;
    1 usage
    public Button callButton;
    public MyDataVH(@NonNull View itemView) {
        super(itemView);
        nameTextBox = itemView.findViewById(R.id.textView);
        callButton = itemView.findViewById(R.id.button);
    }
}
```

Step 5: Create Adapter – MyDataAdapter.java





- Create an adapter class that extends RecyclerView.Adapter<MyDataVH>. This adapter will manage the data and bind it to the views.
- IDE HACK: When you extending a class – use quick fix to implement methods defined by the super class

```
public class MyDataAdapter extends RecyclerView.Adapter<MyDataVH> {

    @NonNull
    @Override
    public MyDataVH onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        return null;
    }

    @Override
    public void onBindViewHolder(@NonNull MyDataVH holder, int position) {

    }

    @Override
    public int getItemCount() {
        return 0;
    }
}
```

MyDataAdapter.java

Step 6: Start by referencing the list of data that will be scrollable

```
public class MyDataAdapter extends RecyclerView.Adapter<MyDataVH> {

    1 usage
    ArrayList<MyData> data;
    public MyDataAdapter(ArrayList<MyData> data){
        this.data = data;
    }
```

Step 7: MyDataAdapter.java – implementing onCreateViewHolder(..)

It's responsible for creating instances of the ViewHolder class, which hold references to the layout views used to display individual items within the RecyclerView.

```
@NonNull
@Override
public MyDataVH onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    LayoutInflater layoutInflater = LayoutInflater.from(parent.getContext());
    View view = layoutInflater.inflate(R.layout.list_item_layout,parent, attachToRoot: false);
    MyDataVH myDataVHolder = new MyDataVH(view);
    return myDataVHolder;
}
```

onCreateViewHolder(@NonNull ViewGroup parent, int viewType)

parent: The ViewGroup that will hold the newly created ViewHolder. This is usually the RecyclerView itself.

viewType: An integer value that can be used to distinguish different types of item views within the adapter. This is particularly useful if your RecyclerView has multiple types of items (e.g., different layouts for different positions).

LayoutInflater layoutInflater = LayoutInflater.from(parent.getContext());

- It obtains a LayoutInflater instance associated with the context of the parent (Activity/Fragment) view (usually the RecyclerView).
- Then, inflate(R.layout.item_layout, parent, false) is used to inflate the viewHolder layout file and create a View object representing a single item in the RecyclerView.

```
@NonNull
@Override
public MyDataVH onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    LayoutInflater layoutInflater = LayoutInflater.from(parent.getContext());
    View view = layoutInflater.inflate(R.layout.list_item_layout,parent, attachToRoot: false);
    MyDataVH myDataVHolder = new MyDataVH(view);
    return myDataVHolder;
}
```

Step 8: MyDataAdapter.java – implementing onBindViewHolder(..)

- It is responsible for taking data from your data source and populating the views within a ViewHolder to display the data for a specific item position in the RecyclerView.

```
@Override
public void onBindViewHolder(@NonNull MyDataVH holder, int position) {
    MyData singleData = data.get(position);
    holder.nameTextBox.setText(singleData.getName());
    holder.callButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast.makeText(view.getContext(), text: "Calling "+ singleData.getPhoneNumber(),
                    Toast.LENGTH_SHORT).show();
            Log.d( tag: "Values", singleData.getPhoneNumber());
        }
    });
}
```

onBindViewHolder(@NonNull MyViewHolder holder, int position)

- holder: The ViewHolder instance that contains the views for the item at the given position.
- position: The position of the item in the dataset that needs to be displayed in this ViewHolder.
- Since the RecyclerView reuses ViewHolder instances as the user scrolls, the onBindViewHolder method is called multiple times with different positions. This allows you to dynamically update the content of views based on the current position.
- If you want to handle clicks on individual items, you can also set up click listeners within the onBindViewHolder method. This way, you can define what happens when an item is clicked.

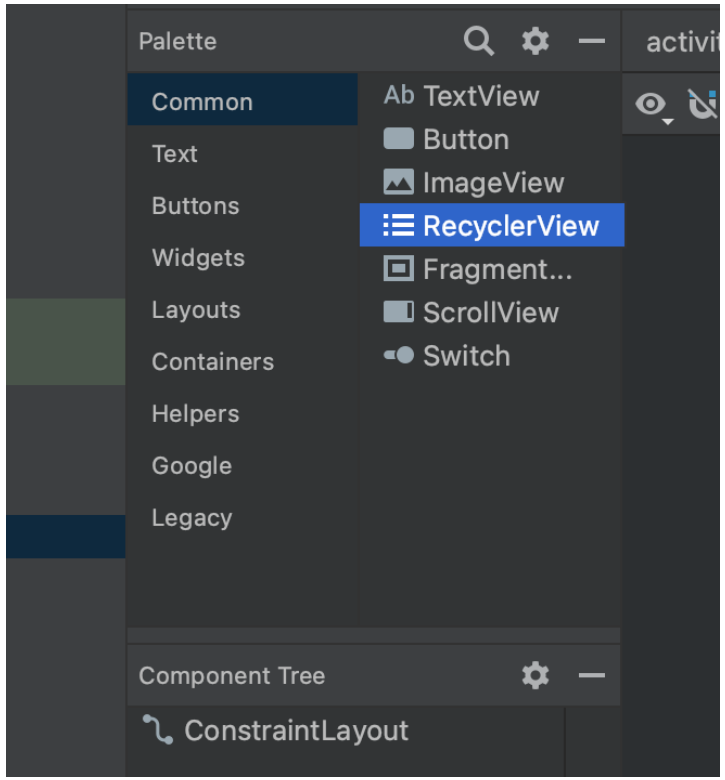
Step 9: implementing getItemCount()

- The main purpose of the getItemCount() method is to provide the total number of items in your data source that the RecyclerView will display.

```
@Override
public int getItemCount() {
    return data.size();
}
```

Step 10: Setup RecyclerView in Activity's layout

In activity.main XML layout, add a RecyclerView element:




```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:a
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 11: Add the data in MainActivity.Java

```
public class MainActivity extends AppCompatActivity {
    18 usages
    ArrayList<MyData> data;
    public MainActivity(){
        data = new ArrayList<MyData>();
        data.add(new MyData( name: "AAAAA", phoneNumber: "0000000"));
        data.add(new MyData( name: "BBBBB", phoneNumber: "000312321"));
        data.add(new MyData( name: "CCCCC", phoneNumber: "054545454"));
        data.add(new MyData( name: "DDDDDD", phoneNumber: "000005"));
        data.add(new MyData( name: "EEEEEE", phoneNumber: "0000050"));
        data.add(new MyData( name: "FFFFFF", phoneNumber: "60000600"));
        data.add(new MyData( name: "GGGGGG", phoneNumber: "60000444"));
        data.add(new MyData( name: "HHHHHH", phoneNumber: "70003333"));
        data.add(new MyData( name: "IIIIII", phoneNumber: "90004444"));
        data.add(new MyData( name: "JJJJJJ", phoneNumber: "80077777"));
        data.add(new MyData( name: "KKKKKK", phoneNumber: "80444444"));
        data.add(new MyData( name: "LLLLLL", phoneNumber: "600054353"));
        data.add(new MyData( name: "MMMMMM", phoneNumber: "5000543545"));
        data.add(new MyData( name: "NNNNNN", phoneNumber: "3000543543"));
        data.add(new MyData( name: "OOOOOO", phoneNumber: "2000545435"));
        data.add(new MyData( name: "PPPPPP", phoneNumber: "1000666666"));
        data.add(new MyData( name: "QQQQQQ", phoneNumber: "343543"));

    }
```

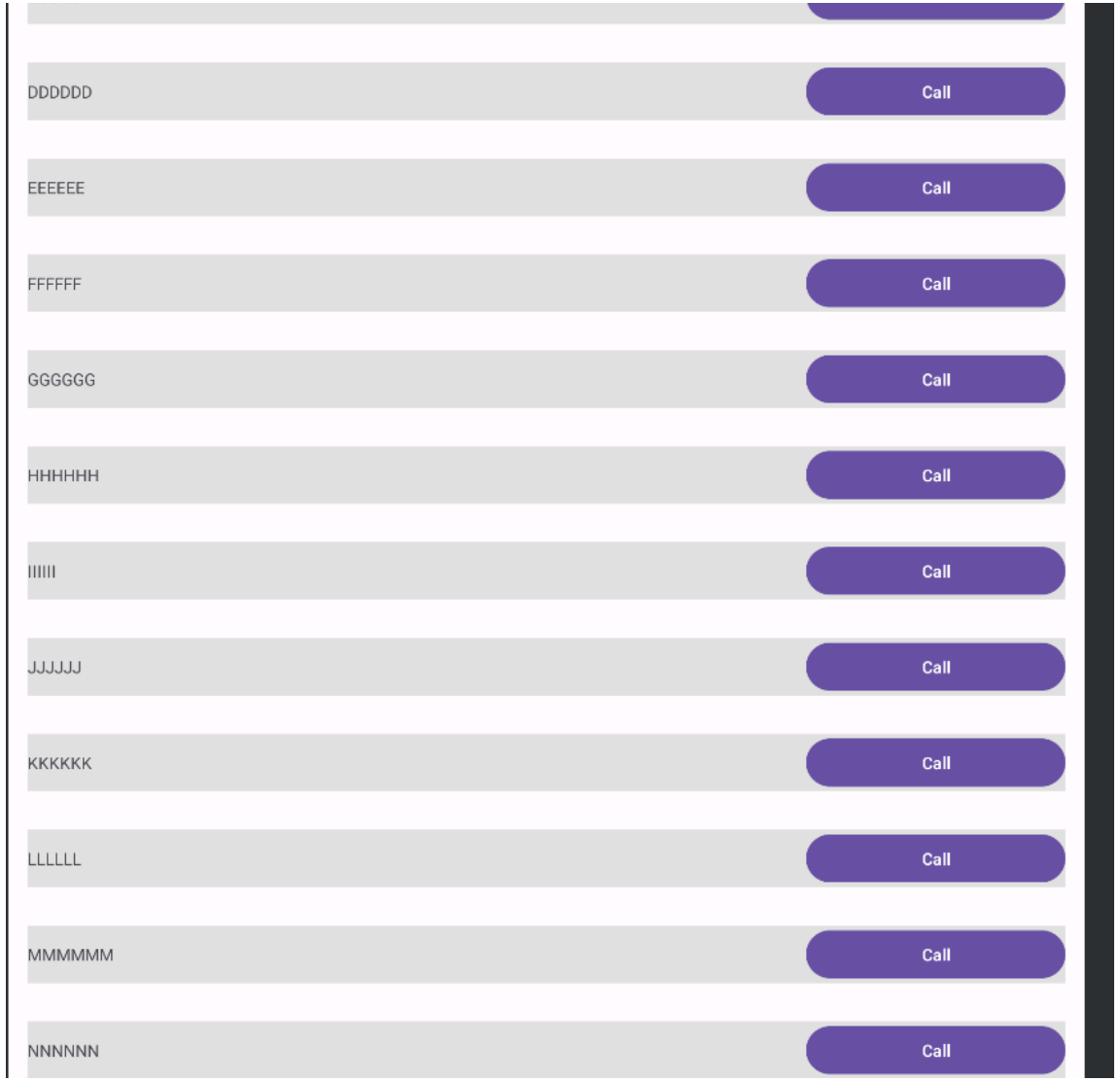
Step 12: Create the view in the MainActivity.java

1. Setting the layout manager for the RecyclerView
2. Creating the Adapter with data
3. Set the adapter to the recyclerview

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    RecyclerView rv= findViewById(R.id.recView);
    rv.setLayoutManager(new LinearLayoutManager( context: this));
    //rv.setLayoutManager(new GridLayoutManager(this,2));
    MyDataAdapter adapter = new MyDataAdapter(data);
    rv.setAdapter(adapter);
}
```

RecyclerView.setLayoutManager()

- A layout manager in the context of a RecyclerView is responsible for measuring and positioning items within the RecyclerView container.
- Different layout managers provide different ways of arranging items, such as in a linear list, grid, staggered grid, or more custom layouts.
- LinearLayoutManager is used as the layout manager. This will arrange the items linearly in either a vertical or horizontal direction, depending on the orientation you specify.



```
RecyclerView.setLayoutManager(new
LinearLayoutManager(this,
LinearLayoutManager.VERTICAL,false));
```

LinearLayoutManager.VERTICAL: This constant indicates that you want the orientation of the layout manager to be vertical.

LinearLayoutManager.HORIZONTAL: This constant indicates that you want the orientation of the layout manager to be horizontal.

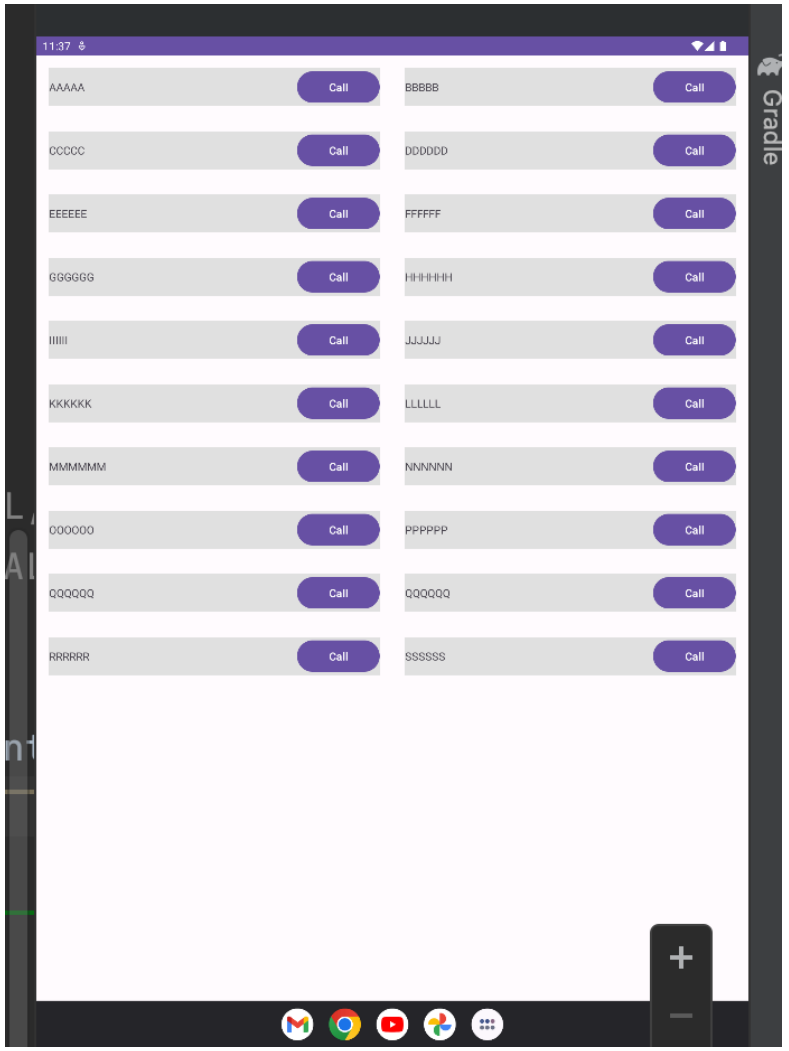
The third argument of the LinearLayoutManager constructor (false in this case) specifies whether the layout manager should reverse the layout. If set to true, the items will be laid out in reverse order.

GridLayoutManager

- The GridLayoutManager provides both vertical and horizontal scrolling depending on how you set up the layout manager.

```
int spanCount = 2;
GridLayoutManager gridLayoutManager = new GridLayoutManager( context: this, spanCount,
        GridLayoutManager.VERTICAL,  reverseLayout: false);
rv.setLayoutManager(gridLayoutManager);
```

- This code sets up a GridLayoutManager with two columns and vertical scrolling. Each row will contain the specified number of items (in this case, two), and the RecyclerView will scroll vertically as needed to accommodate the items.
- Use GridLayoutManager.HORIZONTAL is used to indicate horizontal scrolling.



GridLayoutManger – dynamically setting width/height in a cell

- In the case of vertical scrolling, the width (number of columns) is set dynamically by the span count. However, we are not sure about the height of the cell.
- But we can specify programmatically.
- First, change the height to “0dp” from the wrap content.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_margin="16dp"
    android:background="#E0E0E0">
```

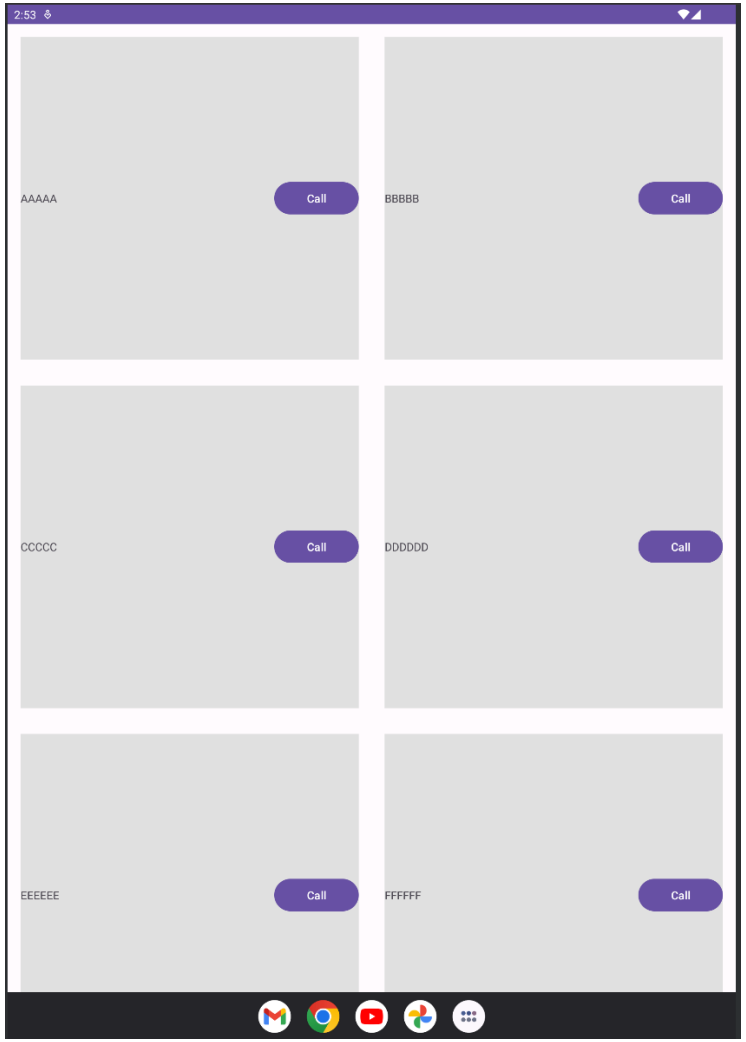
GridLayoutManger – dynamically setting width/height in a cell

- In the ViewHolder Set the height programmatically.

```
public class MyDataVHAdv extends RecyclerView.ViewHolder{
    2 usages
    public TextView nameTextBox;
    2 usages
    public Button callButton;
    1 usage
    public MyDataVHAdv(@NonNull View itemView, ViewGroup parent) {
        super(itemView);
        int hSize = parent.getMeasuredHeight() /3;
        ViewGroup.LayoutParams lp = itemView.getLayoutParams();
        lp.height = hSize;
        nameTextBox = itemView.findViewById(R.id.textView);
        callButton = itemView.findViewById(R.id.button);
    }
}
```

- The parent layout is passed from the adapter

```
public MyDataVHAdv onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    LayoutInflater layoutInflater = LayoutInflater.from(parent.getContext());
    View view = layoutInflater.inflate(R.layout.list_item_layout_adv,
            parent, attachToRoot: false);
    MyDataVHAdv myDataVHolderAdv = new MyDataVHAdv(view,parent);
    return myDataVHolderAdv;
}
```



GridLayoutManger – dynamically setting width/height in a cell

- In the ViewHolder Set the width programmatically. (Required for horizontal scrolling)

```
public MyDataVHAdv(@NonNull View itemView, ViewGroup parent) {
    super(itemView);
    int hSize = parent.getMeasuredHeight() /3;
    int wSize = parent.getMeasuredWidth()/2;
    ViewGroup.LayoutParams lp = itemView.getLayoutParams();
    lp.height = hSize;
    lp.width = wSize;
    nameTextBox = itemView.findViewById(R.id.textView);
    callButton = itemView.findViewById(R.id.button);
}
```

- Questions ??