Buse Çarık 16/07/18

INTERRA R&D Material Design

Material Design

RecyclerView

https://github.com/InterraMaterialDesign/RecyclerView

**INDEX**

1. **INTRODUCTION…………………………………………………….…………………3**
   1. **Purpose of this document……………………………………………………….….3**
   2. **Overview…………………………………………………………………………….3**
2. **WHAT IS RECYCLERVIEW….………………………………………………………4**
   1. **Terms……...……………………...………………………………………………….4**
   2. **ViewHolder………………………………………………………………………….5**
   3. **Adapter………………………………………………………………………………6**
   4. **LayoutManager……………………………………………………………………...8**
   5. **ItemAnimator.……………….……………………………………………….…….12**
   6. **ItemDecoration……………………………………………….……………….……12**
   7. **Nested Classes………………………………………………………………………13**
   8. **ItemTouchHelper………………………………………………….…….…………16**
   9. **Multiple ViewHolders………………………….……….…….……………………21**
3. **ADVANTAGES AND DISADVANTAGES……….……….…….…………...………27**
4. **REQUIREMENTS FOR IMPLEMENTATION….……………………….………....28**
5. **AREAS OF USAGE…………………………………………………………………….28**
6. **REFERENCES……………………………………………………………………….…30**

**INTRODUCTION**

**PURPOSE OF THIS DOCUMENT**

The aim of this project is analyzing RecyclerView.

**OVERVIEW**

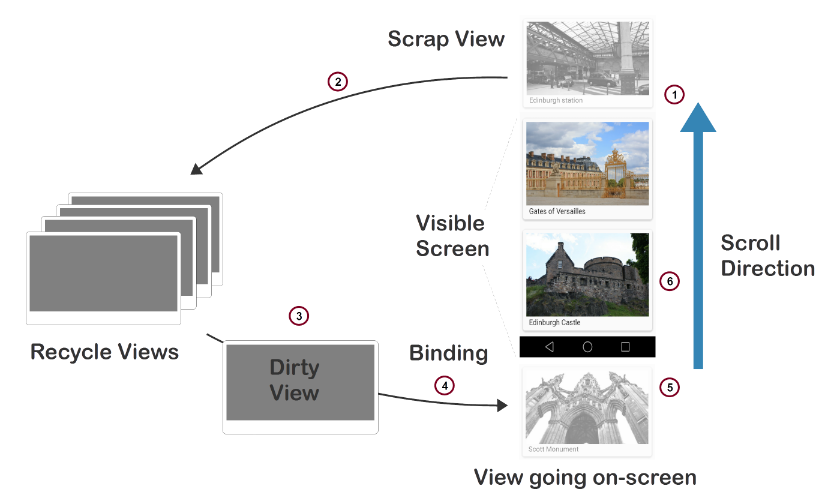
The project that is described in this document, examine the Recycler View’s description, its comparison with ListView, where it is used and its relationship with Card View.

**RecyclerView**

RecyclerView is used for displaying large data set in a limited window. It extends from ***View Group***. It is developed as an alternative to ***ListView*** and ***GridView***. It is ideal for lists with have large number of data and similar content. In RecyclerView, instead of creating a new view for each item in the data set, it produces only limited amount of views which are re-used by the RecyclerView. This saves the memory and increases the performance. Before the details, there are some terms from android developer:

**Terms**

* ***Adapter:*** manages the data model and provides views for the items in the data set.
* ***Position:*** item’s position in an Adapter.
* ***Index:*** the index of an attached child view.
* ***Binding:*** in binding process; a child view is used to display item corresponding to position within the adapter.
* ***Recycle (view):*** re-using view that was used before for a specific position in an adapter for the same type of data.
* ***Scrap (view):*** it is the child view which detaches temporarily during layout. These views can be reused without completely detached from the parent RecyclerView.
* ***Dirty (view):*** it is thechild view which the adapter rebinds before to display.



There are three main parts that need to be known to work with RecyclerView. These are ***RecyclerView.Adapter,*  *RecyclerView.LayoutManager*** and ***RecyclerView.ItemAnimator.*** It needs ***RecyclerView.Adapter*** and ***RecyclerView.LayoutManager*** to be instantiated. Adapter manages the data model and provides views for them. LayoutManager positions the views in the RecyclerView. ItemAnimator animates the views that are changed, added or deleted.

**ViewHolder**

ViewHolder is used as static inner class in RecyclerView.Adapter. It extends from RecyclerView.ViewHolder. ViewHolders are managed by the Adapter. The ViewHolder objects represents the views in each row. It stores references to the views for one entry in the RecyclerView. It holds references to the UI components views inside the tag field of the layout with ***findViewById()*** method. This pattern increases the performance by preventing unnecessary ***findViewById()*** calls on each bind. RecyclerView creates the required number of ViewHolders. That number is usually the number of views on the screen, 2 above of the screen and 2 below of the screen views.

class MovieViewHolder extends RecyclerView.ViewHolder {

TextView movieName;

ImageView movieImageView;

TextView movieRate;

MovieViewHolder (View itemView) {

super(itemView);

movieImageView = itemView.findViewById(R.id.movieImage);

movieName = itemView.findViewById(R.id.movieName);

movieRate = itemView.findViewById(R.id.movieRate);

}

}

**Adapter**

An adapter is extended from ***RecyclerView.Adapter<CustomAdapter.CustomViewHolder>*** and it sets to the RecyclerView with ***setAdapter()*** method. Adapter provides data to the RecyclerView and manages the ViewHolder. The main difference between the classic adapter and RecyclerView.Adapter is that RecyclerView.Adapter is forced to implement ViewHolder; but in regular adapter, the ViewHolder is optional. RecyclerView.Adapter creates ViewHolder as needed and binds the data and the viewholders each other. To implement a RecyclerView.Adapter, first you create a ViewHolder, then you must override three functions: ***onCreateViewHolder***, ***onBindViewHolder*** and ***getItemCount***.

public class MovieAdapter extends RecyclerView.Adapter<MovieAdapter.MovieViewHolder> {

private ArrayList<Movie> movieArrayList;

private Context context;

MovieAdapter(Context context, ArrayList<Movie> movieArrayList) {

this.context = context;

this.movieArrayList = movieArrayList;

}

**,**

* **onCreateViewHolder:** This method is called only when a new view is created. This is invoked by the LayoutManager. Basically, it creates a new view and set it to the layout by inflating an XML layout file. Then, it returns a new viewholder object. This new holder object is used to reach the view in the inflated layout. The layout that is used here, is designed for the data model.

@Override

public MovieViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.grid\_movie, parent, false);

return new MovieViewHolder(view);

}

* **onBindViewHolder:** One of the main responsibilities of the adapter is binding the data to the viewholder. This method binds the viewholder to its data. ViewHolder’s position is passed to the RecyclerView. It uses the data corresponding to that position in the list. This method fills the views in the viewholder’s layout with the data.

However, RecyclerView does not call this method when the position of the views changes on the screen. This is one of the advantages of the RecyclerView. It calls this function when a new viewholder is create or an existing viewholder is reused. When the user scrolls the screen, only the new comer view is bind, the rest stays same, only their layout position is changed. Also, when an item in the data set is deleted, added or changed, again only the viewholder at that position is reconnected. When you make a change in the data set, you must notify the adapter with ***notifyItemRemoved(int position) / notifyItemInserted(int) / notifyItemChanged(int)*** methods to rebinds that item. If you use ***notifyDataSetChanged()*** method instead of these specific notifications, the adapter rebinds all the viewholders on the screen.

@Override

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

Movie movies = movieArrayList.get(position);

holder.movieName.setText(movies.getMovieName());

holder.movieRate.setText(movies.getMovieRate());

holder.movieImageView.setImageResource(movies.getMovieImage());

}

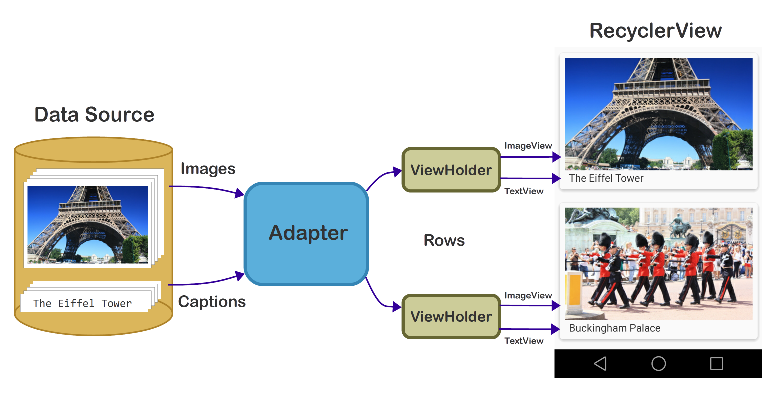
* **getItemCount:** This method returns the total number of items in the data set.

@Override

public int getItemCount() {

return movieArrayList.size();

}



**LayoutManager**

RecyclerView is not interested in positioning views in the layout file. LayoutManager, the subclass of RecyclerView is responsible of this. LayoutManager calculates the layout, positions each view on the screen and determine when to reuse used views that do not appear on the screen. To reuse these views, LayoutManager asks the adapter with ***getViewForPosition()*** method to re-bind the view with a new item from data set. Thanks to viewholder pattern, in that process, you do not need to call findViewById() method.

RecyclerView provides built-in layout managers which are ***LinearLayoutManager*** (both horizontally and vertically), ***GridLayoutManager*** and ***StaggeredLayoutManager***. Or you can create your own custom layout manager by extending ***RecyclerView.LayoutManager***.

* ***LinearLayoutManager***: displays the items in a horizontal or vertical list.
* ***GridLayoutManager***: displays the items in grid view.
* ***StaggeredLayoutManager***: displays the items like in grid view but each column slightly offset from the one before.

There are two ways to connect Layout Manager and RecyclerView. Firstly, you can handle this dynamically with:

RecyclerView.LayoutManager mLayoutManager = new LinearLayoutManager(this, LinearLayoutManager.HORIZONTAL, false);  
recyclerView.setLayoutManager(mLayoutManager);

Orientation reverse layout Context

RecyclerView.LayoutManager mLayoutManager = new GridLayoutManager(this, 2);  
recyclerView.setLayoutManager(mLayoutManager);

Context Span Count

RecyclerView.LayoutManager mLayoutManager = new StaggeredGridLayoutManager(2, StaggeredGridLayoutManager.VERTICAL);  
recyclerView.setLayoutManager(mLayoutManager);

Span Count Orientation

Other option: you can set the LayoutManager in XML file.

<**android.support.v7.widget.RecyclerView  
 android:id="@+id/recycler"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:orientation="vertical"**

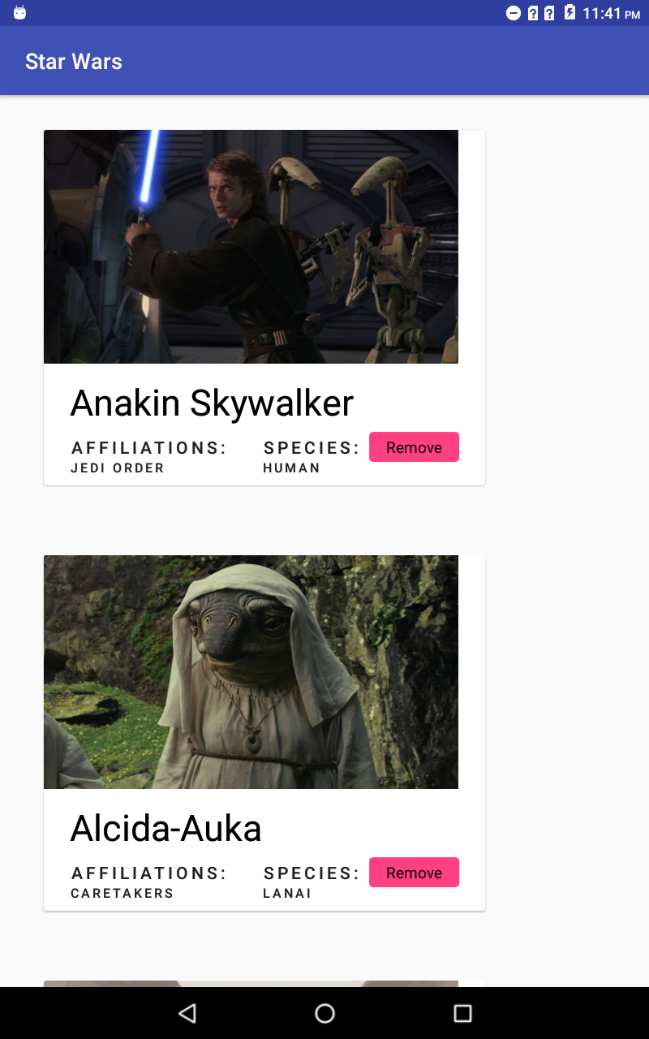
**app:spanCount="2"  
 app:stackFromEnd="true"  
 app:reverseLayout="true"  
 android:scrollbars="vertical"  
 app:layoutManager="****android.support.v7.widget.LinearLayoutManager"**>  
</**android.support.v7.widget.RecyclerView**>

***layoutManager***: set the manager. android.support.v7.widget.LinearLayoutManager, android.support.v7.widget.GridLayoutManager and android.support.v7.widget.StaggeredGridLayoutManager.

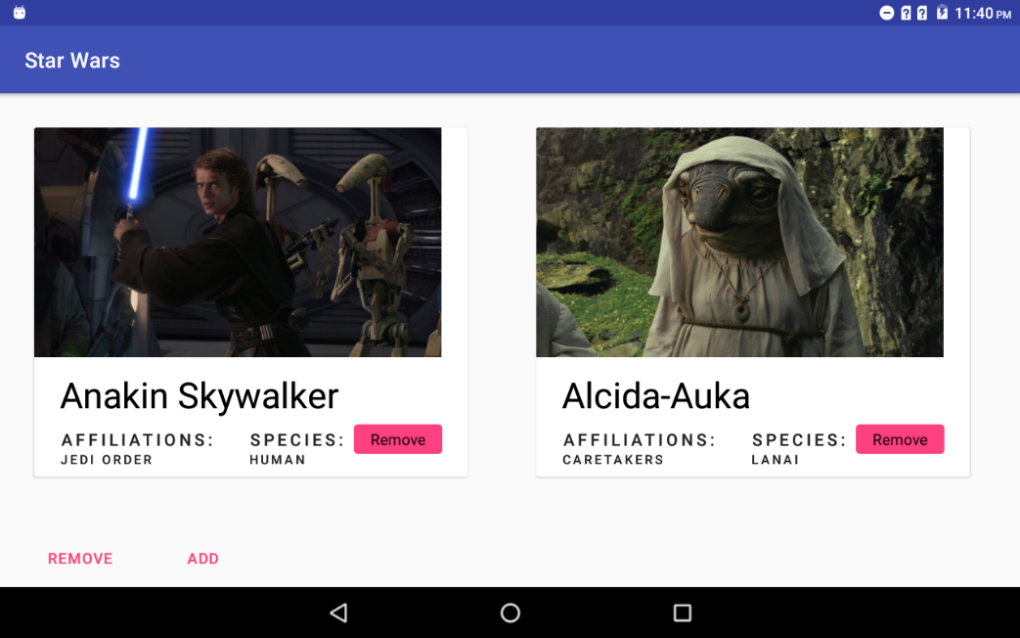
***spanCount***: number of columns.

***stackFromEnd***: start to display from the end of the list.

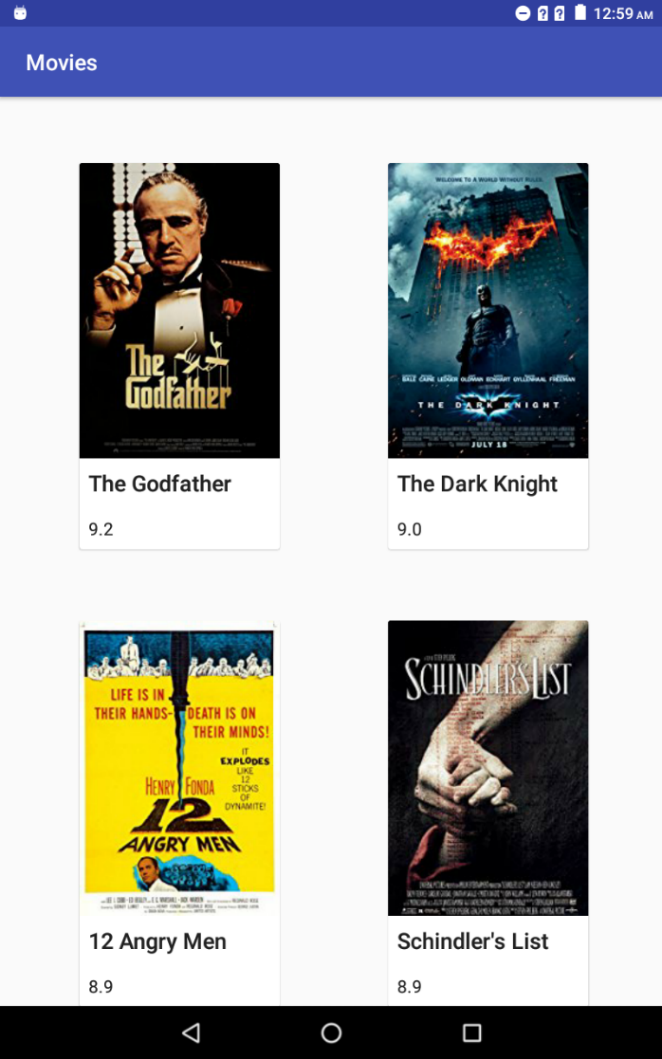
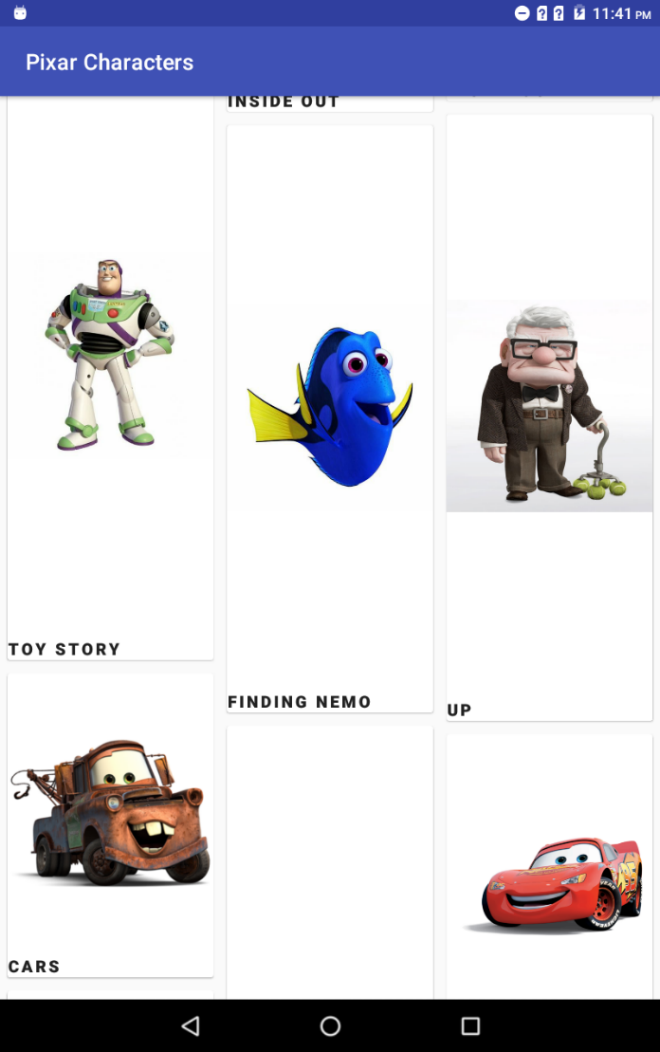
***reverseLayout***: display the list in reverse order.



*LinearLayoutManager.VERTICAL*



*LinarLayoutManager.HORIZONTAL*

*GridLayoutManager StaggeredLayoutManager*

**ItemAnimator**

RecyclerView provides animation with ***RecyclerView.ItemAnimator*** to the views when a changing made on an item is notified the adapter. ***DefaultItemAnimator*** that is used by RecyclerView by default, provides basic animation for add/remove/move events. Default animations can be changed with Item Animator’s methods like ***setRemoveDuration()*** or ***endAnimation().*** Also, you can use ItemAnimator class to create custom animation. If you create your own custom ItemAnimator object or you modify the DefaultItemAnimator, you should connect it with RecyclerView. To set the item animator object to the recycler view, use ***setItemAnimator()*** method.

To observe animation, you should notify the adapter. But if you use ***notifyDataSetChanged()*** method, the animation is not worked because this causes to re-bind all the views without any animation. Therefore, when an item is changed, you should notify the adapter with specific notifications.

RecyclerView.ItemAnimator itemAnimator = **new** DefaultItemAnimator();

itemAnimator.setRemoveDuration(300);  
itemAnimator.setAddDuration(3000);  
itemAnimator.setMoveDuration(1000);

recyclerView.setItemAnimator(ItemAnimator);

**ItemDecoration**

ItemDecoration adds special drawing and offset for each item views and allows them to be distinguished from each other. The use of ItemDecoration can change according to the type of created list. In list views, it provides divider with ***DividerItemDecoration*** object to separate item views like in the ListView. This object can be set as both vertical and horizontal. However, when CardView is preferred for each item, there is no need to use divider.

To use that default divider, first define an item decoration object and add it to the your RecyclerView:

RecyclerView.ItemDecoration ItemDecoration = **new** DividerItemDecoration(**this**, DividerItemDecoration.***VERTICAL***);

recyclerView.addItemDecoration(ItemDecoration);

You can add many ItemDecorations for a RecyclerView. You can also apply this ItemDecorations to specific index. When you add item decoration object to the RecyclerView, ***addItemDecoration()*** method takes two parameters; ***ItemDecoration*** object and int ***index***.

ItemDecoration abstract class has 3 methods. These are: ***onDraw***(Canvas c, RecyclerView parent), ***onDrawOver***(Canvas c, RV parent) and ***getItemOffsets***(Rect outRect, int itemPosition, RV parent). Contents drawn with ***onDraw*** method are drawn before the item views; therefore, they locate under the views. In ***onDrawOver*** method, contents are drawn after the item views hence, they locate top of the views. ***getItemOffset*** method is called by LayoutManager to calculates the size of each item views.

**Nested Classes**

RecyclerView has nested classes other than above.

* **RecyclerView.AdapterDataObserver**: it observes changes in RecyclerView.Adapter class. It has 5 method: ***onChanged***(int positionStart, int itemCount), ***onItemRangeChanged***(int positionStart, int itemCount), ***onItemRangeInserted***(int positionStart, int itemCount), ***onItemRangeMoved***(int fromPosition, int toPosition, int itemCount) and ***onItemRangeRemoved***(int positionStart, int itemCount).
* **RecyclerView.ChildDrawingOrderCallback**: it is used for changing drawing order of the item views. It is used with ***getChildDrawingOrder***(int childCount , int i) (i = current iteration) method. It has ***onGetChildDrawingOrder***(int childCount, int i) that returns the index of the child view to draw for this iteration.
* **RecyclerView.EdgeEffectFactory**: it provides to customize scroll edge effect.
* **RecyclerView.LayoutParams**: it extends ViewGroup.MarginLayoutParams. This is subclass of child views of RecyclerView. It has 4 constructors; RecyclerView.LayoutParams(Context c, AttributeSet attrs)

RecyclerView.LayoutParams(int width, int height)

RecyclerView.LayoutParams(ViewGroup.MarginLayoutParams source)

RecyclerView.LayoutParams(ViewGroup.LayoutParams source)

RecyclerView.LayoutParams(RecyclerView.LayoutParams source). It has ***getViewAdapterPosition()*** method to gets “the up-to-date adapter position that the view this LayoutParams is attached to corresponds to” and getViewLayoutPosition() method to gets “the adapter position that the view this LayoutParams is attached to corresponds to as of latest layout calculation”. It has also ***isItemChanged(),*** ***isItemRemoved()***, ***isViewInvalid()*** and ***viewNeedsUpdate()*** methods.

* **RecyclerView.OnChildAttachStateChangeListener**: it notifies whenever ViewHolder is attached or detached from RecyclerView. It has two methods: ***onChildViewAttachedToWindow***(View view) and ***onChildViewDetachedFromWindow***(View view)
* **RecyclerView.onFlingListener**: it is used for implementing custom fling behavior. boolean ***onFling***(int velocityX, int velocityY) method.

* **RecyclerView.onScrollListener**: it can be added to the RecyclerView to get knowledge when the scrolling is happened. It has two callback method: ***onScrollStateChanged***(RecyclerView recyclerView, int newState) and ***onScrolled***(RecyclerView recyclerView, int dx, int dy). ***newState*** gets one of the constant scrolled state which are ***SCROLL\_STATE\_IDLE, SCROLL\_STATE\_DRAGGING*** or ***SCROLL\_STATE\_SETTLING***. dx is the amount of horizontal scroll and dy is the amount of vertical scroll.
* **RecyclerView.RecycledViewPool**: if you create one, RecyclerView provides one. This class enables to share views between RecyclerViews. To use same pool, first create a RecycledViewPool instance and set to the other RecyclerView with ***setRecycledViewPool***(RecycledViewPool). With that method, you can re-use a pool for multiple RecyclerViews. Methods:

***getRecycledView*** (int viewType); if that type of viewholder is present in the pool, returns that viewholder otherwise return null.

***putRecycledView*** (RecyclerView.ViewHolder scrap); if the pool is full for that type of ViewHolder, it will be scrapped. scrap is the viewholder that is added to the pool. ***clear***(); it discards all the viewholders. These 3 methods are enabled to manipulate the pool’s content.

***getRecycledViewCount*** (int viewType); it returns the current number of views of that viewType that held by the pool.

***setMaxRecycledViews*** (int viewType, int max); it sets the maximum number of viewholders to hold in the pool before discarding.

* **RecyclerView.Recycler**: it controls the scrapped or detached views for reuse. It has ***bindViewToPosition***(View view, int position) to bind the view to the position. ***recycleView***(View view) method which “recycles a detached view”. ***setViewCacheSize***(int viewCount) which sets the maximum number of detached views that are retained for reuse. ***getViewForPosition***(int position) which “obtains view initialized for the given position”. ***getScrapList*** () which “returns the list of viewholders that are currently in the scrap list”. ***convertPreLayoutPositionToPostLayout*** (int position) method can be used for converting the pre-layout position to the post-layout position.
* **RecyclerView.OnItemTouchListener**: In RecyclerView, there is no ***OnItemClickListener()*** or ***OnItemLongClickListener()*** methods. Instead, ***OnItemTouchListener()*** can be used with gesture detector. “An OnItemTouchListener allows the application to intercept touch events in progress at the view hierarchy level of the RecyclerView before those touch events are considered for RecyclerView's own scrolling behavior”. This class It has 3 methods: ***onInterceptTouchEvent***(RecyclerView rv, MotionEvent e); it observes silently touch events before RecyclerView or its child views process them. ***onRequestDisallowInterceptTouchEvent***(boolean disallowIntercept); it is called when a touch event is not wanted to be prevented with ***onInterceptTouchEvent*** method. ***onTouchEvent***(RecyclerView rv, MotionEvent e); “Process a touch event as part of a gesture that was claimed by returning true from a previous call to onInterceptTouchEvent”.
* **RecyclerView.State**: it contains information about RecyclerView’s current state. “If you implement custom components, you can use State's ***put/get/remove*** methods to pass data between your components without needing to manage their lifecycles.”.
* **RecyclerView.ViewCacheExtension**: it is a helper class. It enables developer to control on view caching. It has ***getViewForPositionAndType***(Recycler, int, int) method. When LayoutManager calls the ***getViewForPosition***(int), recycler first look at scrap view and first level cache to find suitable view. If it cannot find, before the pool, it calls the ***getViewForPositionAndType***(Recycler, int, int) method. This method does not create a new view. It returns previously created reusable view for that given position and type.

More detailed information about ***ViewCacheExtension*** can be found in that link: <https://android.jlelse.eu/anatomy-of-recyclerview-part-1-a-search-for-a-viewholder-continued-d81c631a2b91>

**ItemTouchHelper**

This is support class to the RecyclerView that adds drag & drop and swipe-to-dismiss movements to the RecyclerView. It needs a callback class and a RecyclerView. It is extended from ItemDecoration class. It can be added easily to the Layout Managers with ***attachToRecyclerView***(RecyclerView recyclerView) method. If you want to use ItemTouchHelper, you must create an ***ItemTouchHelper.Callback*** which allows to control touch event like swipe and drag. It also provides the callback to override animations.

ItemTouchHelper.Callback callback = new ItemTouchHelper.Callback() {

@Override

public int getMovementFlags(@NonNull RecyclerView recyclerView, @NonNull RecyclerView.ViewHolder viewHolder) {

return 0;

}

@Override

public boolean onMove(@NonNull RecyclerView recyclerView, @NonNull RecyclerView.ViewHolder viewHolder, @NonNull RecyclerView.ViewHolder target) {

return false;

}

@Override

public void onSwiped(@NonNull RecyclerView.ViewHolder viewHolder, int direction) {

}

};

ItemTouchHelper itemTouchHelper = new ItemTouchHelper(callback);

itemTouchHelper.attachToRecyclerView(recyclerList);

When you create an ItemTouchHelper.Callback object, you should override ***getMovementFlags()***, ***onMove()*** and ***onSwiped()*** methods.

* **getMovementFlags()** method enables to specify which directions you swipe and drag. makeMovementFlags() builds returned flags. In our example, we enable both up and down drag and left-right swipe.

@Override

public int getMovementFlags(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder) {

return makeMovementFlags(ItemTouchHelper.UP | ItemTouchHelper.DOWN, ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT);

}

Instead of ItemTouchHelper.Callback, you can use ***ItemTouchHelper.SimpleCallback*** which is helper to use basic implementation. It is a wrapper to default Callback that construct with directions of swipe and drag.

ItemTouchHelper.Callback callback = new ItemTouchHelper.SimpleCallback(ItemTouchHelper.UP | ItemTouchHelper.DOWN, ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT) {

@Override

public boolean onMove(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, RecyclerView.ViewHolder target){

return false;

}

@Override

public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction){

}

};

* **onMove()** method is overridden if you want to make any changings during the drag movement. To drag an item, you should change this item’s place from old position to new position in your adapter and must call ***notifyItemMoved***(int fromPosition, int toPosition) method. Note: if you want to control where the item is dragged, ***canDropOver***(RecyclerView, ViewHolder, ViewHolder) method can be overridden.
* **onSwiped()** method is called when an item is swiped. The animation is animated until the view goes beyond the limit of the screen. After that, this method is called. In that method, you should update your adapter and call proper ***notify…()*** method to inform the adapter. As an example, we delete the item from the list:

@Override

public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction) {

int position = viewHolder.getAdapterPosition();

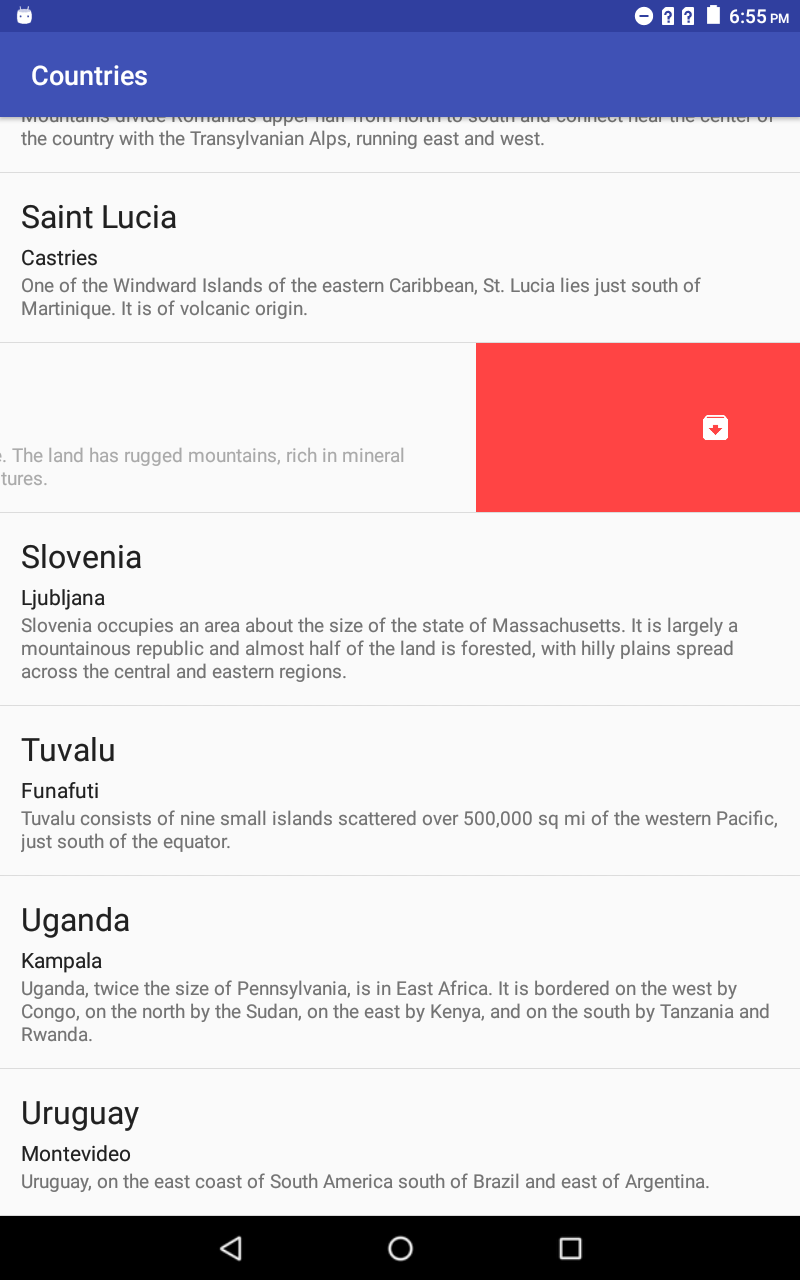
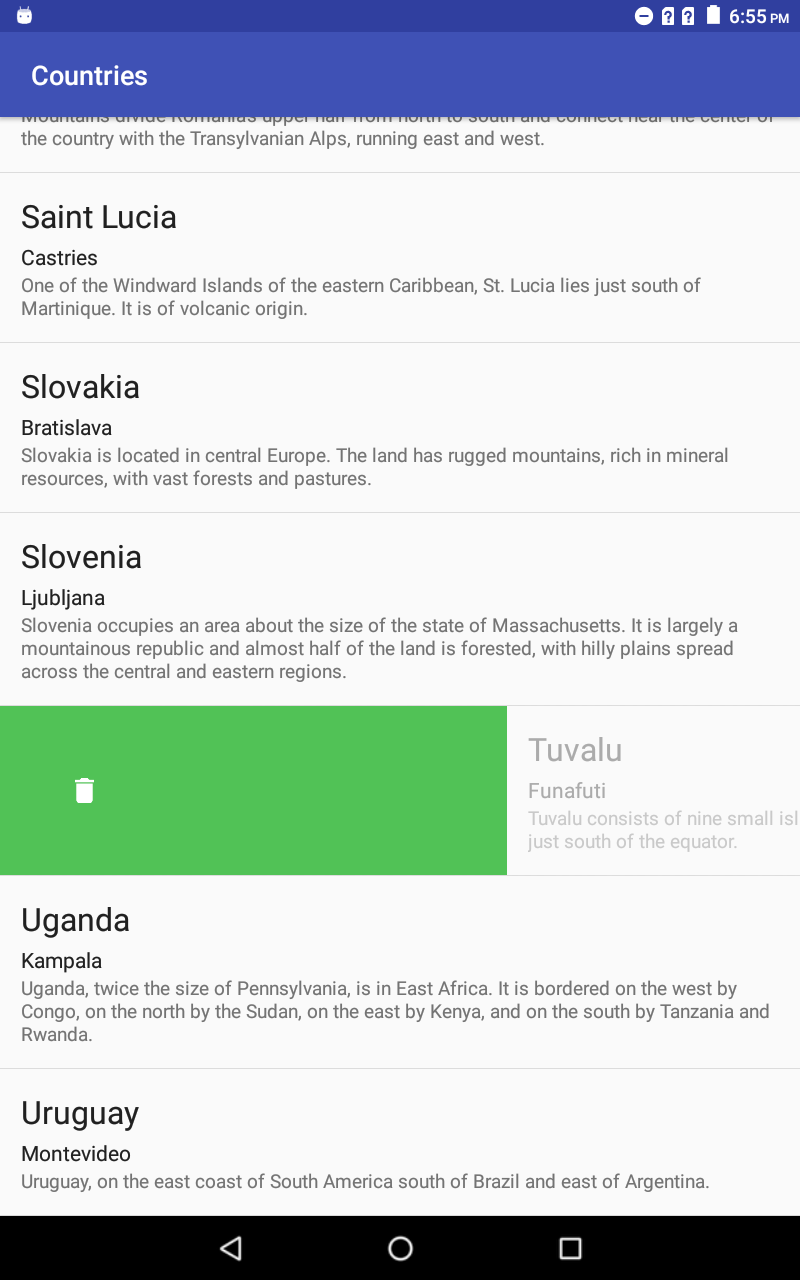
mList.remove(position);

adapter.notifyItemRemoved(position);

}

You can use ItemTouchHelper.Callback to drag without swipe actions or vice-versa. These options are set with ***isItemViewSwipeEnabled()*** and ***isLongPressDragEnabled()*** method.

The other important method is **onChildDraw**(Canvas c, RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, float dX, float dY, int actionState, boolean isCurrentlyActive). To customize the view’s appearance according to respond of user’ actions, this method should be overridden.

For that appearance, you can override ***onChildDraw()*** method like this:

@Override

public void onChildDraw(Canvas c, RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, float dX, float dY, int actionState, boolean isCurrentlyActive) {

Bitmap icon;

if (actionState == ItemTouchHelper.ACTION\_STATE\_SWIPE) {

View view = viewHolder.itemView;

Paint paint = new Paint();

float height = view.getBottom() - view.getTop();

float width = height/3;

//right

if (dX < 0) {

paint.setColor(getResources().getColor(R.color.red));

RectF background = new RectF((float) view.getRight() + dX, (float)view.getTop(), (float) view.getRight(), (float) view.getBottom());

c.drawRect(background, paint);

icon = BitmapFactory.decodeResource(getResources(), R.mipmap.archive\_foreground);

RectF icon\_dest = new RectF((float)view.getRight() - 2\*width, (float) view.getTop() + width, (float)view.getRight() - width, (float) view.getBottom() - width);

c.drawBitmap(icon, null, icon\_dest, paint);

}

//left

else {

paint.setColor(getResources().getColor(R.color.green));

RectF background = new RectF((float) view.getLeft(), (float)view.getTop(), dX, view.getBottom());

c.drawRect(background, paint);

icon = BitmapFactory.decodeResource(getResources(), R.mipmap.delete\_image\_foreground);

RectF icon\_dest = new RectF((float) view.getLeft() + width, (float) view.getTop() + width, (float) view.getLeft() + 2\*width, (float) view.getBottom() - width);

c.drawBitmap(icon, null, icon\_dest, paint);

}

final float alpha = 1.0f - Math.abs(dX) / (float) viewHolder.itemView.getWidth();

viewHolder.itemView.setAlpha(alpha);

viewHolder.itemView.setTranslationX(dX);

}

else {

super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive);

}

}

Other methods:

* **chooseDropTarget**(RecyclerView.ViewHolder selected, List<RecyclerView.ViewHolder> dropTargets, int curX, int curY): selects drop target from the list of the ViewHolders.
* **clearView**(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder): is called the user’s interactions is over and animation is finished.
* **convertToAbsoluteDirection**(int flags, int layoutDirection): converts a given set of flags to absolution.
* **convertToRelativeDirection**(int flags, int layoutDirection): replaces a movement direction with its relative version by taking layout direction into account.
* **getAnimationDuration**(RecyclerView recyclerView, int animationType, float animateDx, float animateDy)
* **getDefaultUIUtil**(): returns the ItemTouchUIUtil that is used by the ItemTouchHelper.Callback class for visual changes on Views in response to user interactions.
* **getMoveThreshold**(RecyclerView.ViewHolder viewHolder): returns the fraction that the user should move the View to be considered as it is dragged.
* **getSwipeEscapeVelocity**(float defaultValue): defines the minimum velocity which will be considered as a swipe action by user.
* **getSwipeThreshold**(RecyclerView.ViewHolder viewHolder): returns the fraction that the user should move the View to be considered as swiped.
* **getSwipeVelocityThreshold**(float defaultValue): defines the maximum velocity ItemTouchHelper will ever calculate for pointer movements.
* **onChildDrawOver**(Canvas c, RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, float dX, float dY, int actionState, boolean isCurrentlyActive)
* **onMoved**(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, int fromPos, RecyclerView.ViewHolder target, int toPos, int x, int y): is called after the onMove() method returns true.
* **onSelectedChanged**(RecyclerView.ViewHolder viewHolder, int actionState): is called when a ViewHolder is dragged or swiped by the ItemTouchHelper.

**Multiple ViewHolders**

A RecyclerView can have multiple ViewHolders. ViewHolder object must be created according to its type. Therefore, ***onCreateViewHolder()*** method takes viewType as parameter. And when searching a ViewHolder in the pool for a view, RecyclerView first learns that view’s view type with ***getItemViewType***(position) method to search according to that type in the pool. In default, this method assumes that there is only one type; therefore, it returns 0. If you use multiple ViewHolders, you should rewrite this method according to your types.

@Override

public int getItemViewType(int position) {

switch (mList.get(position).getType()) {

case 0:

return Model.FIRST\_TYPE;

case 1:

return Model.SECOND\_TYPE;

default:

return -1;

}

}

Another point is that each view type has its own pool of ViewHolders and has its own capacity. This number is 5 by default. It cannot retain more than 5 viewholder but this number can be changed by ***setMaxRecycledViews***(int viewType, int capacity) method. Through this method, memory can be used efficiently. As an example, in a RecyclerView that has different ViewHolder types, if one of the types has large number of views on the screen, you should increase the capacity of the pool for that view type. For another type, if it does not display on the screen more than one, setting its pool capacity as 1 prevents the memory from being wasted.

When creating a RecyclerView with multiple ViewHolders, you should start by adding the type to the model. While creating adapter for multiple ViewHolders, you should not extend it from RecyclerView.ViewHolder. onCreateViewHolder() method has a viewType parameter to decide which type of ViewHolder is created.

public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

View view;

switch (viewType) {

case Model.SECOND\_TYPE:

view = LayoutInflater.from(parent.getContext()).inflate(R.layout.view\_holder1, parent, false);

return new viewHolderType1(view);

case Model.FIRST\_TYPE:

view = LayoutInflater.from(parent.getContext()).inflate(R.layout.view\_holder2, parent, false);

return new ViewHolder2(view);

}

return null;

}

In onBindViewHolder() method, contents are bind to the ViewHolder according to the type of ViewHolder.

public void onBindViewHolder(@NonNull final RecyclerView.ViewHolder holder, final int position) {

Model character = mList.get(position);

if (character != null) {

switch (character.getType()) {

case Model.SECOND\_TYPE:

((viewHolder1) holder).mName.setText(character.getmName());

((viewHolder2) holder).mSpecies.setText(character.getmSpecies());

break;

case Model.FIRST\_TYPE:

((viewHolder2) holder).cName.setText(character.getmName());

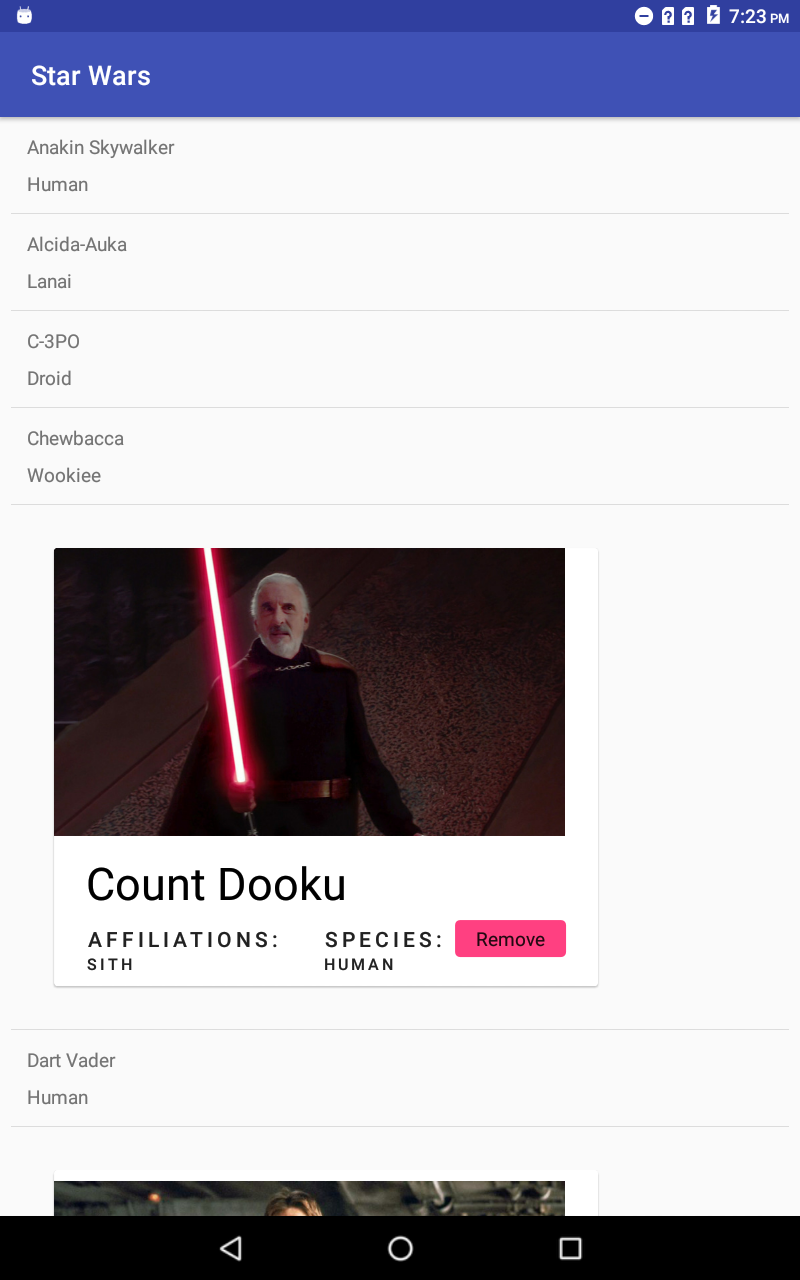
((viewHolder2) holder).cSpecies.setText(character.getmSpecies());

break;

}

}

}



1 2

Layout Manager

Cache

RV first check cache

getViewForPosition()

Cache= yes

3

**RecyclerView**

getViewHolderByType()

Recycled Pool

Cache=no

getItemViewType()

3 5 4

onBindViewHolder()

onCreateViewHolder()

5

gVHBT() = yes

gVHBT()

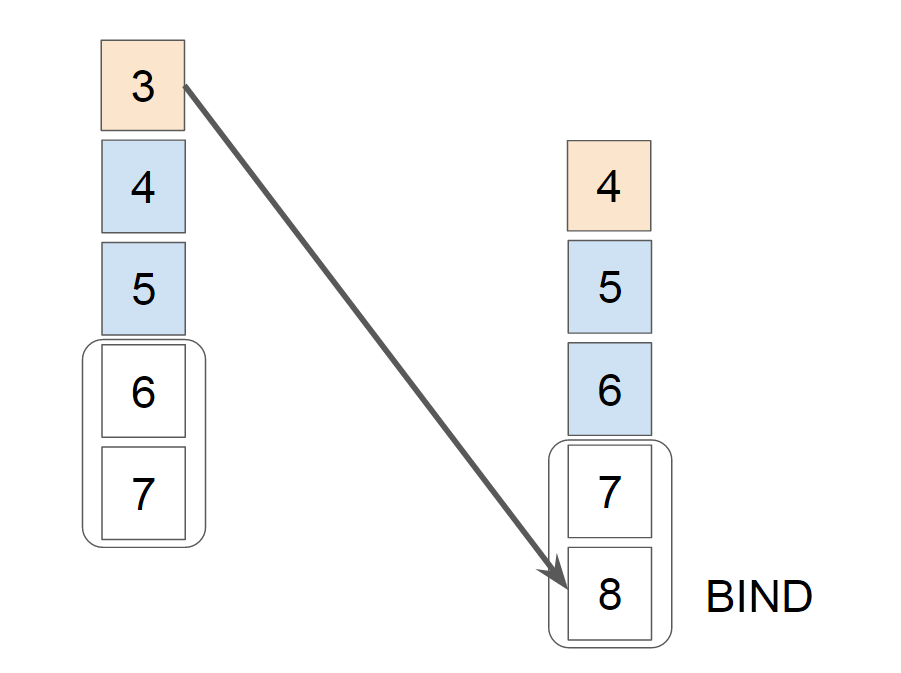
= no

Return viewType

Adapter

While Layout Manager is calculating the layout, it wants view with getViewForPosition() method from the RecyclerView for position 6 for example. Then RecyclerView first check cache views for position 6. Cache is like an ArrayList of ViewHolders and it is not separated according to the view type. By default, it is capacity is 2 but this number can be increased with ***setItemViewCacheSize()*** method. If the view is found in cache views, it will return it to the Layout Manager. Otherwise, if the view cannot be found in the cache views, RecyclerView asks the view type to the Adapter with ***getItemViewType()***. Then, RecyclerView turns and checks Recycled.Pool for this view type with ***getViewHolderByType()*** method. It checks whether there is a ViewHolder for this type in the pool. If there is not, RecyclerView returns to the Adapter and Adapter creates a new ViewHolder with ***onCreateViewHolder()*** method and binds it. If it has one, Adapter only binds the position with this ViewHolder. In a nutshell; “

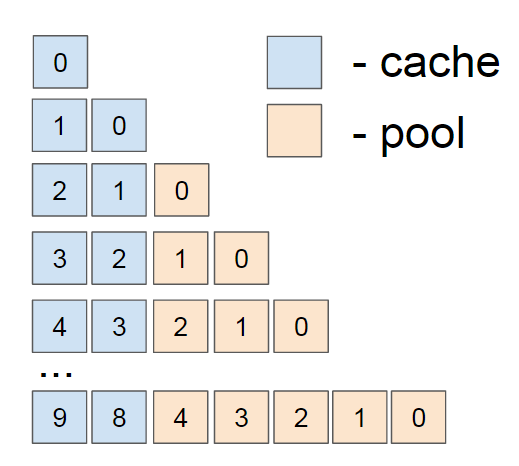
* If a ViewHolder was found nowhere, it is created and bound.
* If a ViewHolder was found in pool, it is bound.
* If a ViewHolder was found in cache, there is nothing to be done.”



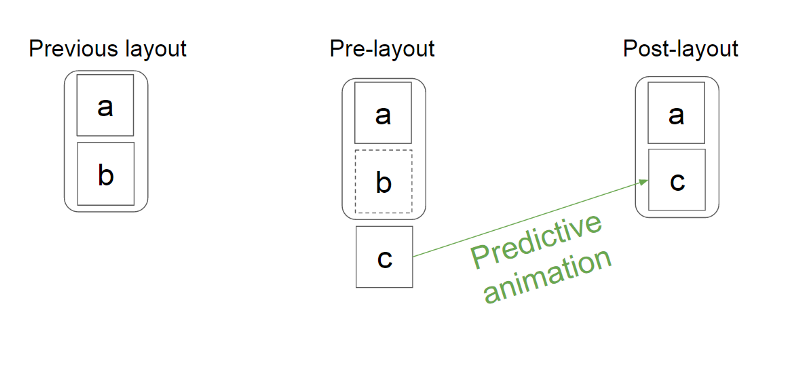
Cache views

Then, Layout Manager adds back this view to the RecyclerView and RecyclerView informs to the Adapter that this view is added to the layout with ***onViewAttachedToWindow()*** callback. When Layout Manager finished working with this view, RecyclerView again informs to the Adapter that this view is not visible in the layout with ***onViewDetachedFromWindow()*** callback.

As a side note; in the Recycled.Pool, dirty views are found, and they require rebinding. The ViewHolders in the pool, only have view type and corresponding view, its states like position are cleared. But when in cache views, it retains some of the states like position. The main difference between ViewHolders in the pool and ViewHolders in the cache views is that when ViewHolders in the pool are being searched according to the view type, the cache views are searched by the given position.



When an item in the list is removed, added or changed, item animator ensures that these actions done smoothly. So, while a changing happening, Layout Manager wants to add/remove and recycle views, but ItemAnimator wants to animate. It is difficult for layout manager to track adapter changes to calculate animations therefore, ChildHelper was created. “ChildHelper provides a virtual children list to the Layout Manager”. When Layout Manager calls getChild(), removeChild() or addChild() methods, ChildHelper creates and uses a pre-layout until item animator finishes its work. The pre-layout uses previous adapter state. Layout Manager does not involve this process. This type of animations is called as ***predictive animations***. Therefore, two types of position are existed, one of them is adapter position and the other one is layout position. These positions are different between time the adapter.notify…() and updated layout is calculating.



**Advantages and Disadvantages**

RecyclerView is created instead of ListView and GridView. Compared to the other two options, it has many advantages. Firstly, ListView supports only vertical linear layout and cannot be customize. But in RecyclerView, LayoutManager supports different types like ***GridLayoutManager***, ***StaggeredGridLayoutManager*** and ***LinearLayoutManager*** (can be horizontal or vertical).

ListView has attribute for dividers between items which is ***android:divider***. But in RecyclerView, there is a class (RecyclerView.ItemDecoration) that enables designing dividers or other objects manually.

ListView does not have any animation. On the other hand, RecyclerView provides animation support for whenever items are added or removed. You can use default animation, or you can customize with ***RecyclerView.ItemAnimation*** class.

RecyclerView forces to implement custom adapter for different data sets but ListView has adapters for different sources like ArrayAdapter for arrays and CursorAdapter for database results.

The key advantage is ***ViewHolder***. RecyclerView enforces the ViewHolder pattern but ListView does not have such a requirement.

Only thing that should be remembered when using RecyclerView is the adapter must be notified when a change is made to the list.

**Requirements for Implementation**

RecyclerView can be found in the *latest support-v7 version*. It is established in Android 5.0 (Lollipop) and added in version 22.1.0. To use RecyclerView, add the following requirement to the build.gradle file in the project:

dependencies {  
    implementation 'com.android.support: recyclerview-v7:28.0.0-alpha1’

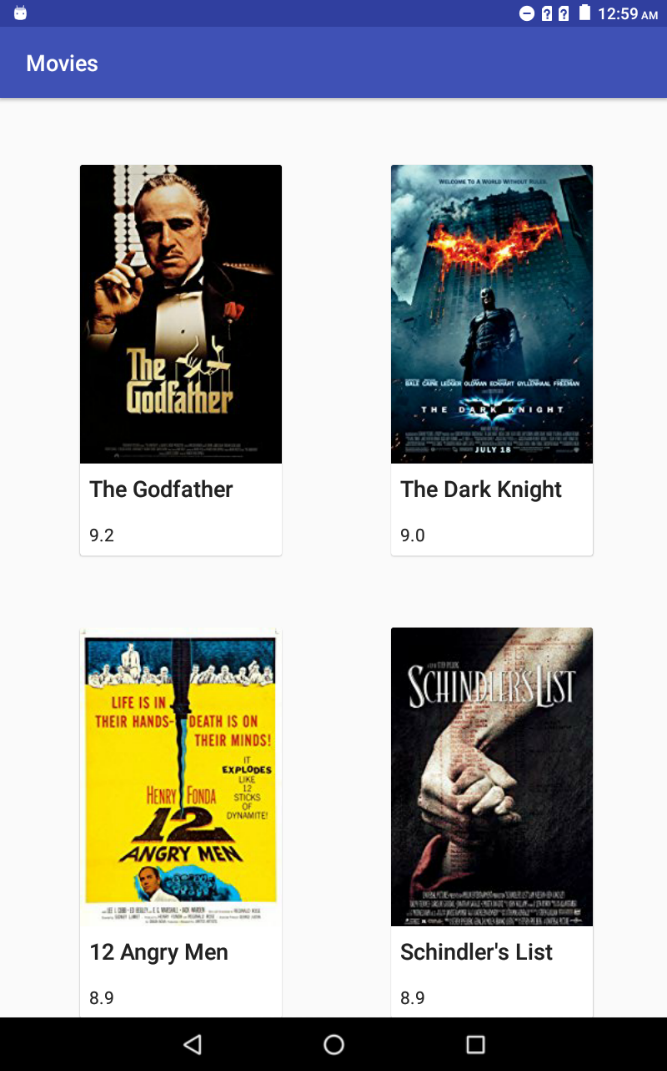
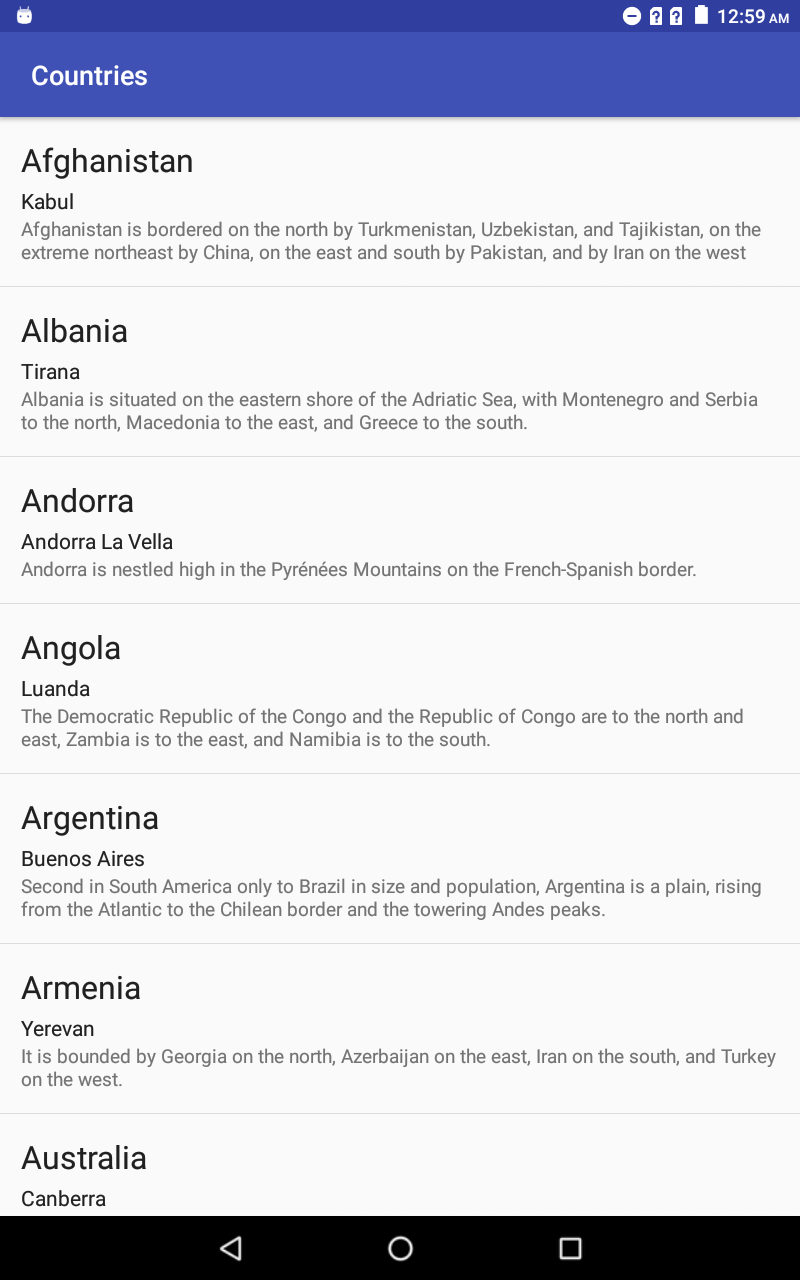
}

In XML file, RecyclerView is called as android.support.v7.widget.RecyclerView. You add the RecyclerView to the layout file which is used by LayoutManager.

<?xml version="1.0" encoding="utf-8"?>  
<android.support.v7.widget.RecyclerView  
    android:id="@+id/my\_recycler\_view"  
    android:scrollbars="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"/>

**Areas of Usage**

RecyclerView is useful for lists that have large number of items that share the same content. RecyclerView is used with CardView or list view. Items that have image and/or nested structure CardView should be used. CardView make it easier to distinguish items from each other because of the rounded corners and elevation effect. The list view should be used in data sets that consist of text views and/or icons that contain general information like contact info. In list view, divider is used to distinguish. RecyclerView is mainly used in listing e-mails, list of products, search result…

CardView List view

**References**

Android Developers. (n.d.). Retrieved June 27, 2018, from https://developer.android.com/

Homepage. (n.d.). Retrieved June 27, 2018, from <https://material.io/>

Movies, TV and Celebrities. (n.d.). Retrieved July 19, 2018, from https://www.imdb.com/

Pixar characters. (n.d.). Retrieved July 19, 2018, from http://disney.wikia.com/wiki/Category:Pixar\_characters

The Official Star Wars Website. (n.d.). Retrieved July 17, 2018, from https://www.starwars.com/

(2017, March 11). Anatomy of RecyclerView: A Search for a ViewHolder – AndroidPub. Retrieved August 1, 2018, from https://android.jlelse.eu/anatomy-of-recyclerview-part-1-a-search-for-a-viewholder-404ba3453714

(n.d.). Retrieved July 17, 2018, from https://www.learnhowtoprogram.com/android/web-service-backends-and-custom-fragments/custom-adapters-with-recyclerview