

Android Building block - Part 2

Download the full App created in this guide

<https://drive.google.com/file/d/1hiATTqbz-KEhcq20dfqWNX4ZEdqG9p3P/view?usp=sharing>

Download the Animations XML Files

<https://drive.google.com/file/d/1lrXBf5dJL7Lu7O3bNNUyHqMBfvYFJLT5/view?usp=sharing>

Single activity architecture

From the Google **I/O 2018**:

“Today we are introducing the Navigation component as a framework for structuring your in-app UI, with a focus on making a single-Activity app the preferred architecture”

Yes, although we can create as many Activities as we want, this not the recommended architecture according to Google. Activity takes the **whole** screen, when creating it the system has to create a new context and switch to it, also it has to create a new window for the Activity's root view and sometimes this can take a while especially if we don't need all of this work.

A long time ago Google introduced a new OS (Android 3.0) just for tablets. The OS included Fragments as her main key feature. Fragments gave us the ability to split our whole screen into a bunch of individual units the can work together and still each one is independent, it has its own Lifecycle events (that corresponds to the hosting Activity lifecycle). Each Fragment has its own Kotlin and XML files and the most important thing is that creating it is much quicker then creating activity since we don't create a new context or an new window for its root view but instead we just add its root view to a specific view container in the Activity layout and it becomes the Fragment host.

Because it was a later addition the Fragments API was added both to the v4 support library (today replaced by the AndroidX) to be used in lower versions of Android and to the android.app package - based on the idea that when enough time will pass we will use only the android.app and won't be needing the support library anymore. But sometimes realty overcomes and today the android.app Fragments are deprecated and we should only use the ones from the AndroidX support library.

Let's go back to the Google I/O, in 2018 they introduced Navigation as a part of the Jetpack tools for clean and reliable android apps. The Navigation component, like the iOS storyboard allows us to create and design in a nice and easy graphical interface all of our app flow in terms of screens and transitions. You can create and see in one place all of your app screens and the flow

between them and it's all done with, what else, Fragments! In fact the only work the Activity is doing is hosting the Fragments and sometimes interacting with app menus.

Fragments

Like said before a fragment has a few important key features: It has its own layout and Kotlin file. This way he his responsible for is own ui and logic. Besides the fact that it makes our code more structural it makes the fragment an individual unit that can be taken to another project with ease. So go ahead and create a new Empty Activity project with an Activity that will use solely asa a container. **This will be an ongoing project so call it ArchitectureProject.**

In it create a new XML file and add a Floating Action Button in the buttom - end of the parent. Your xml should look like this:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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=".ItemsFragment">

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/floatingActionButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:srcCompat="@android:drawable/ic_input_add" />

</androidx.constraintlayout.widget.ConstraintLayout>
```


If you're there already, create our second screen, it will use for adding an item that will be shown later on in a list in that our screen so just add the item input fields in the next screen. Each Item will have a title, a description and an image. So go ahead and create your UI, don't forget the finish button. This is the general layout of the xml file.

Enter title

Enter Description

PICK IMAGE

FINISH



Before we need to create our Fragments, let's understand it's lifecycle:

The lifecycle of the Activity in which the fragment resides directly affects the lifecycle of the Fragment. Each lifecycle callback of the activity results in a similar callback for each hosted Fragment. For example, when the activity receives `onPause()`, each fragment in the activity receives `onPause()`.

Fragments have a few extra lifecycle callbacks that handle unique interaction with the activity in order to perform actions such as build and destroy the fragment's UI. These additional callback methods are:

`onAttach()` - Called when the fragment has been associated with the activity (the `Activity` is passed in here by the OS). If the Fragment needs the Context after this function he can retrieve it using the `getActivity` or `requiredActivity`

functions.

onCreateView() -Called to create the view hierarchy associated with the fragment.

onViewCreated() - Called immediately after onCreateView. This gives subclasses a chance to initialize themselves once they know their view hierarchy has been completely created. The views aren't attached to their parents yet.

onActivityCreated() -Called when the activity's **onCreate()** method has returned.

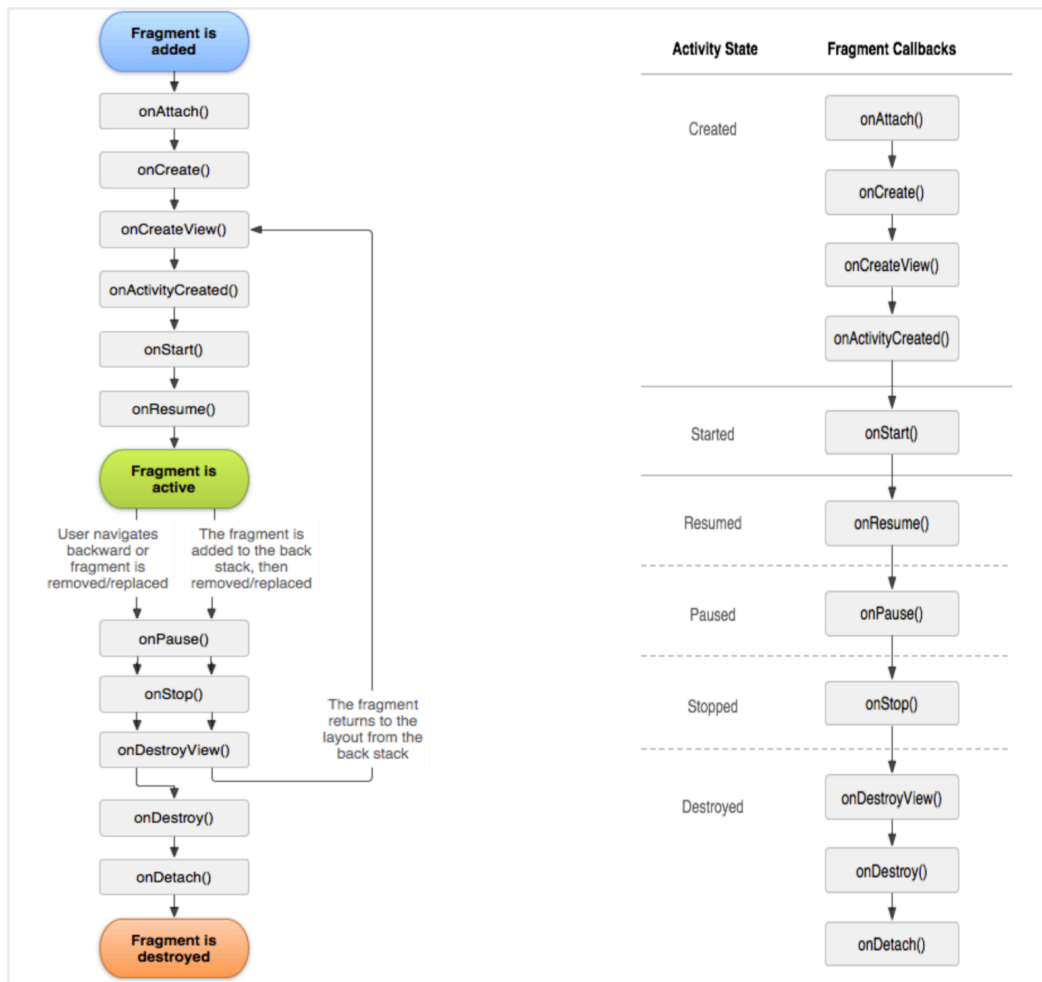
onDestroyView() - Called when the view hierarchy associated with the fragment is being removed.

onDetach() - Called when the fragment is being disassociated from the activity. the getActivity() function here will returned null.

Once the activity reaches the resumed state, you can freely add and remove fragments to the activity. Thus, only while the activity is in the resumed state can the lifecycle of a fragment change independently.

More than that the Fragment's Views has a separate Lifecycle that is managed independently from that of the fragment's Lifecycle.

The fragment views can be destroyed while the fragment itself is alive in the back stack. The back stack designed to imitate the back pressed activity action for fragments - meaning when the user presses the back button the last performed action is popped out. If the action included replacing Fragment A with B then pressing the back button will pop it out and Fragment B will be replaced with A that waited in back stack to be popped out (the instance remained alive while the views weren't). This is important and has affects on the view bidding as we will see soon.



Fragments and the Fragment Manager

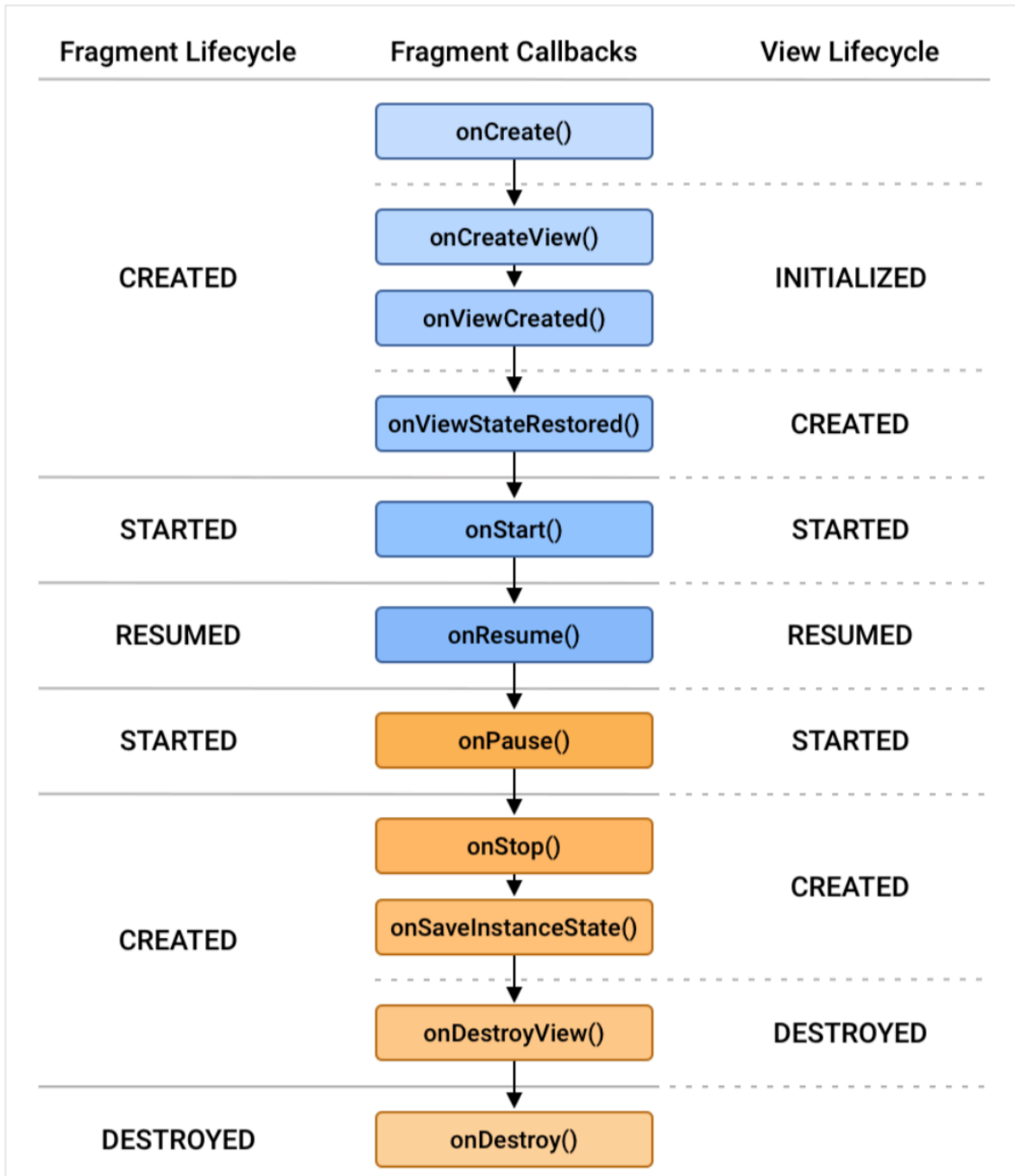
After the **onCreate()** event the fragment is added to the **FragmentManager**. The **FragmentManager** is responsible of attaching fragments to their hosting activity and detaching them. When these events happen the fragments **onAttach()** and **onDetach()** are called. After **onAttach** you can call the **FragmentManager**'s **findFragmentById()** function and get the desired fragment. Besides managing all or our fragments and giving as the ability to add, remove, replace and retrieve them, the **FragmentManager** also manages the back stack we have talked about before.

Like the Activity the Fragment has its own lifecycle and it implements the Jetpack's **LifecycleOwner** interface that allows to retrieve his lifecycle events using the **getLifecycle()** method. This function return a **Lifecycle** object with the following states:

- **INITIALIZED**
- **CREATED**
- **STARTED**
- **RESUMED**
- **DESTROYED**

But don't forget that the fragments keeps a separate lifecycle object for its views in case we need to preform UI related tasks such as start observing data that will only be shown in a list.

Here are the fragment lifecycle events and its view lifecycle events with their corresponding callbacks:



We will see more on those Lifecycle states later on when we dive deeper into Jetpack.

For further reading on the fragment lifecycle please refer to:

<https://developer.android.com/guide/fragments/lifecycle>

Creating our Fragment Kotlin file

First add the viewBinding feature to you app Gradle file:

```
buildFeatures {
    viewBinding true
}
```

Now create your Fragments. Inherit from the AndroidX Fragment class and use view binding to inflate our views. Because the views has a separate lifecycle from the fragment itself and it can outlive its views in the back stack, we need to de-allocate our biding object in the **onDestroyView()** method.

For this we have to make a nullable binding field, initiate it in the **onCreateView()** function where we get the layout inflater and the parent, which serves as the fragment container, and after inflating the layout we return the root view. The binding must be assigned null in the **onDestroyView** which causes the GC to de-allocate all the views and release the memory even if the fragment itself is still alive and in this way we can avoid memory leaks. Please note that because it is nullable we create a non-nullable property for easy access which we will use in caution.

```
class ItemsFragment : Fragment() {

    private var _binding : AllItemsFragmentBinding? = null

    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = AllItemsFragmentBinding.inflate(inflater,container, attachToParent: false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Note that when we inflated the Activity layout we didn't supply any parent because the system create a new window just for it, so it's not joining any parent. But here we specify a container since we add its root view to a specific container resides in the hosting activity.

Before going forward to our Navigation component please add the tools:Context to each fragment's xml file and reference the Kotlin in order for

the android studio Design to show us our views related to this act fragment.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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=".ItemsFragment">
```

Adding Navigation for Fragment transactions

Like said before the Fragment Manager is responsible for exchanging and managing Fragments. Each transition can include adding, removing or replacing fragments and is called **Fragment transaction**. In order to imitate the back button press for fragment as it is with Activity (remove the last added screen) a special back stack is created and you can add the Transaction to it. When the user press the back button the last transaction is removed. The fragment can live in the back stack although it's views are destroyed like we said.

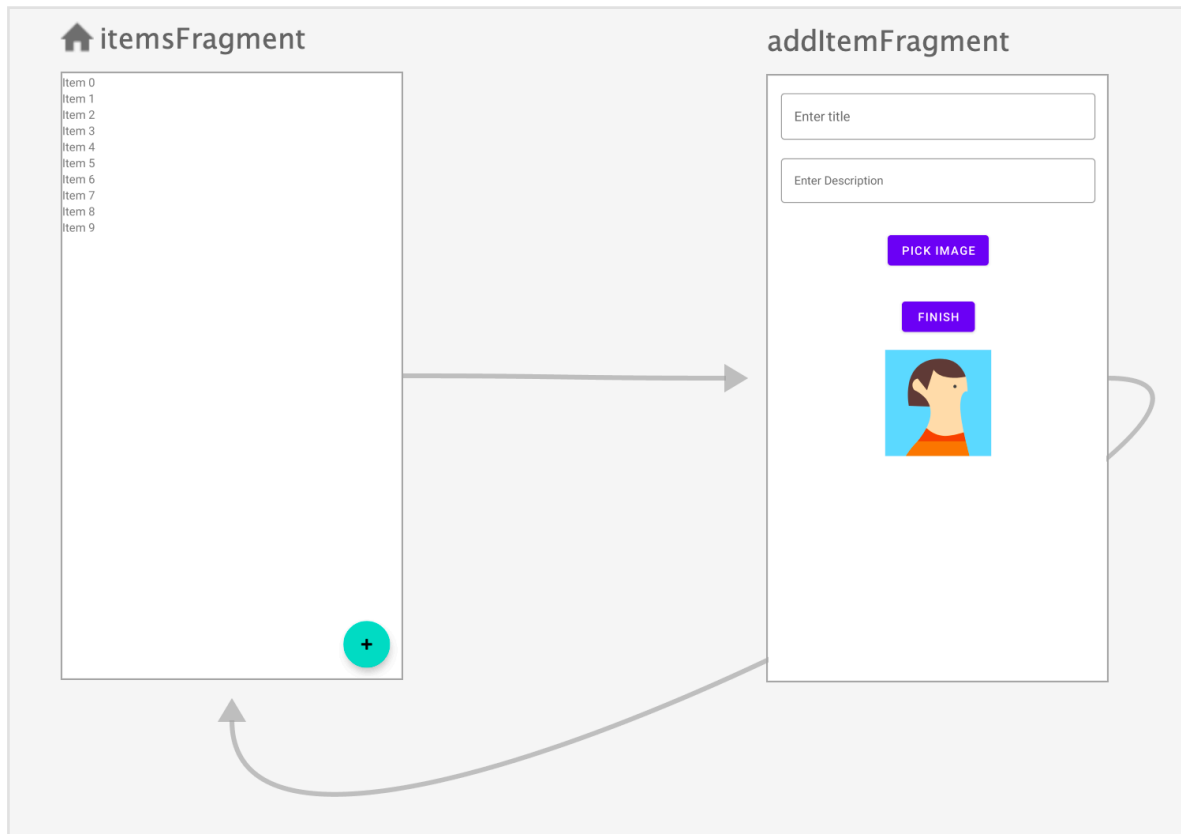
All of this work used to be done manually but as of Google I/O 2018 we can do all of this with the Navigation component.

First we need to add the Navigation graph to our resources and the fragments and their transitions to it. By looking at the graph we will see all of our app screens and the flow between them. We can design our app flow in a very nice and friendly GUI interface and we can even add animations.

So first, implement the following steps:

- Under the res-> new resource. In the dialog choose type Navigation and give it a name. This is your Navigation graph.
- Enter your newly created Navigation xml file and add your fragments. The first fragment you will add will be your home fragment(can be change later on by right clicking on any fragment and setting as home)
- If you can't see the layout in the preview copy the tools from the activity_main.xml file into your navigation xml file and add tools:layout to each fragment and reference his xml file (if you added the tools:context to your root layout of each xml file all should be ok).
- In the design add your actions by dragging from one fragment to another, each arrow added can be executed in our Kotlin code later on - note the addition the the xml file

In the end it will look like that:



By pressing the floating button we will navigate to the adding screen and by pressing the finish button we will go back to our items list.

Next we need need to add the NavHost to your activity.

The navigation host is an empty container where destinations are swapped in and out as a user navigates through your app. When we want to perform our actions we will get a reference to it and execute them. The Navigation host is a simple Layout which inherits from the reliable `FrameLayout` and called **FragmentContainerView**. This Fragment container can create our fragments and execute out fragment transactions. Add it via xml to your root layout - general activity_main.xml file.

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/my_nav" />
```

The **defaultNavHost** property tells the system to pass the back clicks to this NavHost so he can pop his back stack.

Optional you can use the "tag" attribute if you want later to reference him using the Fragment Manager `findFragmentByTag()`.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    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.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/my_nav" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

So now all we have to do is execute the action and pass extra information with some of them.

To perform these actions we need to get a reference to our Navigation controller.

We can do this with any child view in the its view hierarchy to by:

```
Navigation.findNavController(v).navigate([Action id])
```

Where v is any view in a view tree that its root is the Navigation Controller, meaning any view from the displayed Fragments.

If you do not have a live view you can also pass the context and a view id:

```
Navigation.findNavController(this,R.id.text_view).navigate([Action id])
```

But the best and shortest is With the navigation-fragment-ktx library (which already added to your Gradle):

```
From fragment: findNavController().navigate([Action id]);
```

```
From activity: findNavController(R.id.text_view).navigate([Action id])
```

And in our case:

```
binding.finishBtn.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment)
}
```

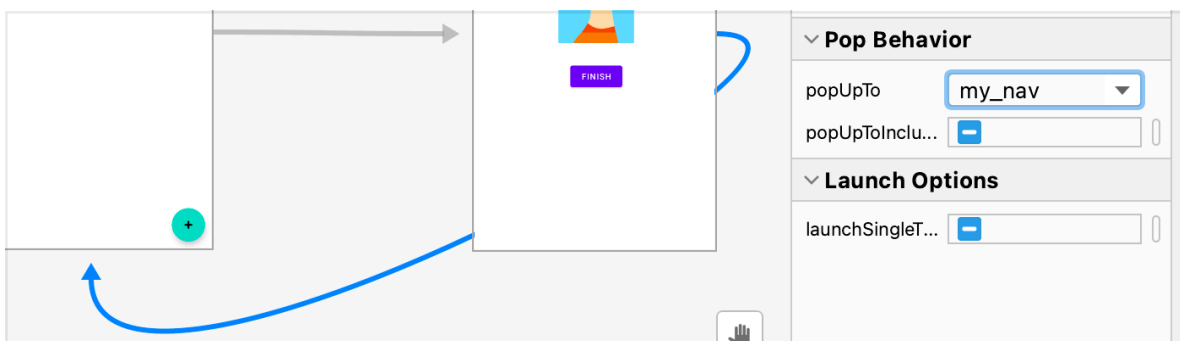
```
binding.floatingActionButton.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_itemsFragment_to_addItemFragment)
}
```

Back stack

Now run the app navigate to the add item screen, press the finish button and go back to the Home Screen. So far so good.

But press the back button. Strange ah? Not so much. By default each action(which is a transaction) is added to the back stack, pressing the back button pop the last action.

You can solve this by pop the back stack with the action. Choose the action In the navigation and in the pop behavior choose the root navigation container. This mean that when executing the action all the fragments in the back stack will get pop up to the very root of the navigation.



Or alternatively you can pop to the home screen but if you do so check the inclusive check box to also pop the former instance of it from thee (otherwise you will have to home screens)



Passing data between Fragments

Each Fragment has an Arguments property which is a bundle and is generally used to pass information to the fragment upon its creation. A common Fragment factory method will receive the data add it to a bundle and set it as the Arguments property of the newly created Fragment it returns. That way this

factory function create a new fragment with the data it needs already inside it. Later on, when we need to get the data it can access its arguments property the get it. The same is done by the our Navigation Controller, when we wants to navigate to a specific fragment and pass some data, we create a bundle and send it with the action to the **navigate()** function. It will automatically set this bundle as the Arguments property of the new Fragment.

Let's pass the item details (not an object and without the photo yet) and show them in a Toast message, for now. Next stage we will create a dynamic list in the all items screen and add the object to it.

Let's create a bundle with the details and call the navigate function with it as a parameter.

```
binding.finishBtn.setOnClickListener { it: View!  
    val bundle = bundleOf( ...pairs: "title" to binding.itemNameEt.text.toString(),  
        "description" to binding.itemDescriptionEt.text.toString())  
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment, bundle)  
}
```

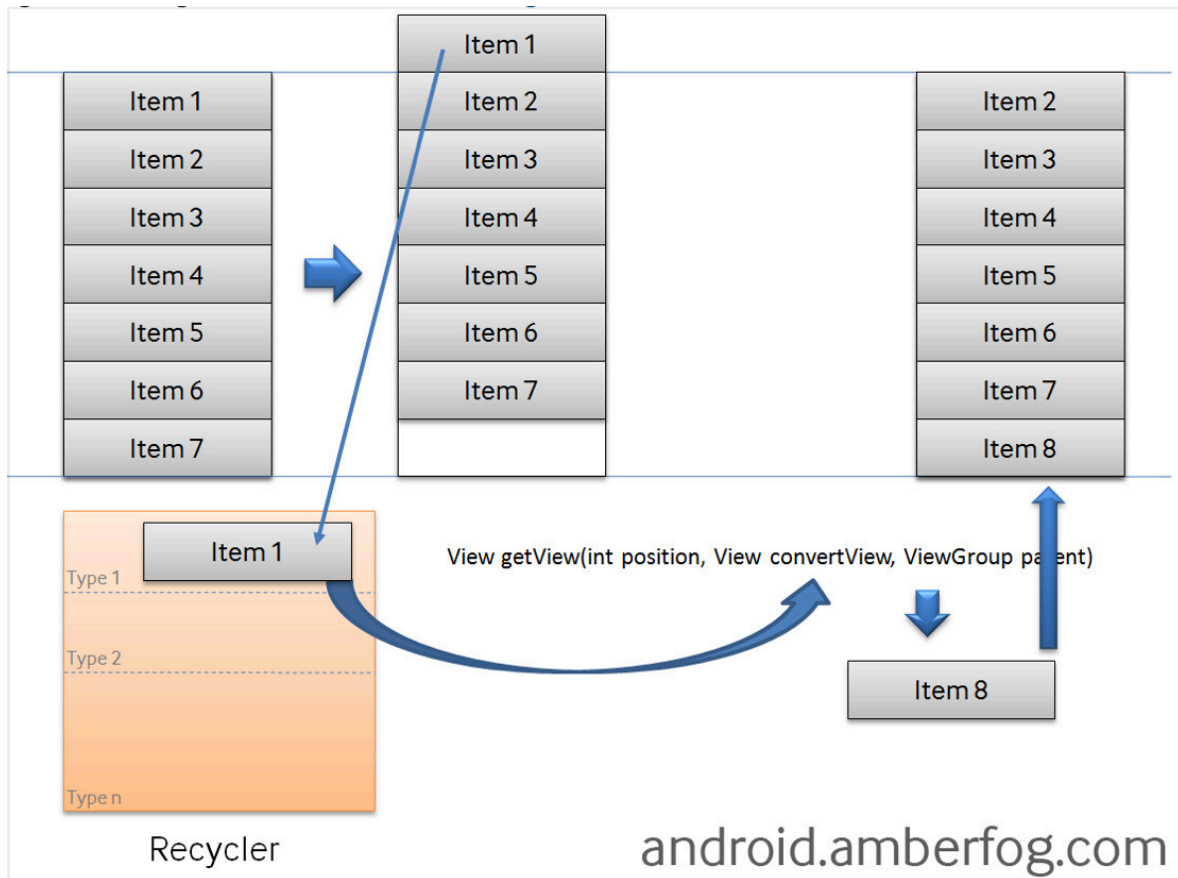
In the newly created fragment that has this bundle in his arguments property we get the data

```
arguments?.getString( key: "title")?.let { it: String  
    Toast.makeText(requireActivity(), it, Toast.LENGTH_LONG).show()  
}
```

RecyclerView & Adapter

A RecyclerView dynamic scrollable list of items. The list is populated dynamically. It is not loading all the cells in advance, but instead it gets them at runtime when the user scroll to it. This is why it uses a Recycle bin like its name suggest. The purpose of the Recycle bin is create a faster scrolling by minimizing the amount of object allocation. In fact the recycler is only creating the amount of initial cells shown to the user and maybe one more. Once the user start scrolling the list the item that is no longer visible is not de-allocated from the heap but instead moves to the recycle bin and when a new cell with the same type (same layout and views) as the old one needs to be created it simply recycle the old one with the new content. This idea is based on the principle on which the content of the cells is different their views isn't so we can simply take an old cell and populate it with the relevant data.

Take a look in the following diagram:



Although what you see here is the old ListView getView() function and the idea is the same.

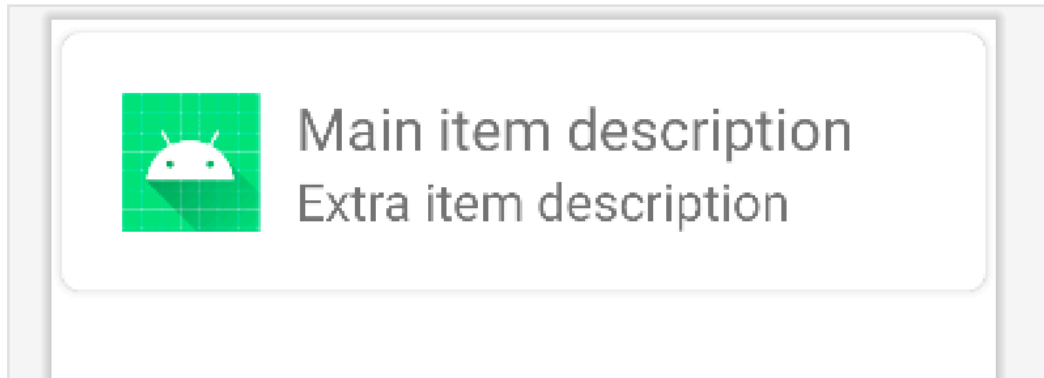
The old item moves to recycle bin and when a new cell should enter if it is from the same type(same layout) as the one in recycle bin the list uses it instead of creating a new one.

So let's use this beautiful mechanism in our project:

First add a RecyclerView to the all_items_layout make it take all the parent space and give it an id

```
<androidx.recyclerview.widget.RecyclerView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/recycler"/>
```

Next design your cell's layout. With RecyclerView we use CardView. So create a new xml with the CardView as the root and design your layout:



To achieve this layout add **contentPadding** and **cornerRadius** while setting the **cardUseCompatPadding** to true in your CardView attributes. This will make nice separation between the cards. Inside the card add an Horizontal linear layout with and image and a vertical linear layout with two TextViews. Don't forget to give each view an id.

Now add the Item data class

```
data class Item(val title:String, val description: String, val photo: String?)
```

Note the the photo property is nullable since not all items will have a photo(at least not in the beginning)

Create an ItemManager object declaration that will serve as a Singleton that holds a list of the items and a functions to add and remove an item to and from the list. Later on we will move the data to the ViewModel and persist it in the local storage with Room database.

So add also this object declaration:

```
object ItemsManger {  
  
    val items : MutableList<Item> = mutableListOf()  
  
    fun add(item: Item) {  
        items.add(item)  
    }  
  
    fun remove(index: Int){  
        items.removeAt(index)  
    }  
}
```

Adapter

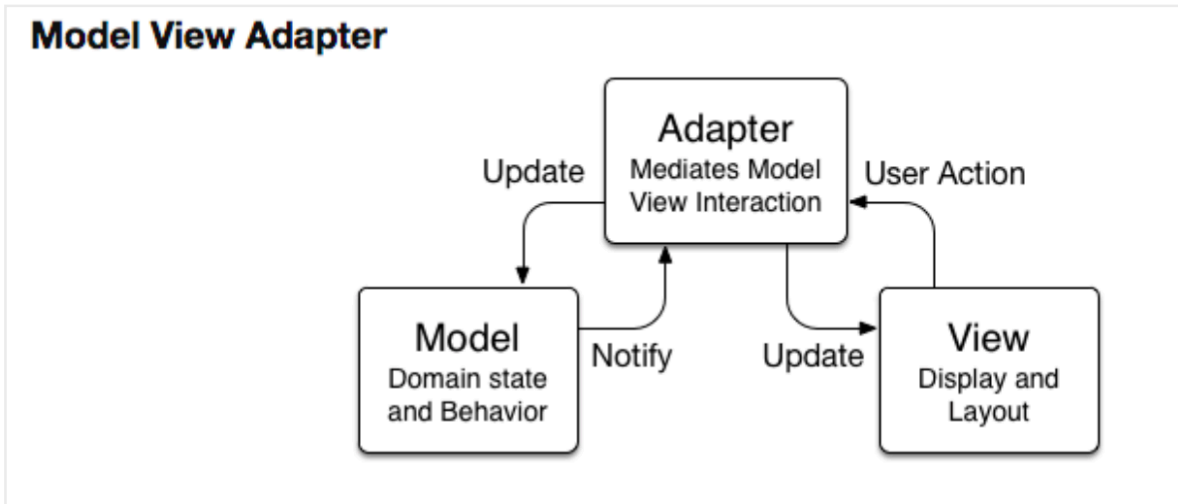
Now that we have both the cell layout the the Kotlin data class we can create an Adapter that connects them together and supply populated views to the recycler.

But First let's understand the concept of the Adapter. According to the MVC design pattern the the controller is a mediator unit between the views and the model in order to separate between the logic and the UI. The MVA (Model View Adapter) is very similar.

Model-View-Adapter is a variation of the Triad where all communication between Model and View must flow through the Adapter, instead of interacting directly as in a Traditional MVC Triad. The Adapter becomes a communication hub, accepting change notifications from Model objects and UI events from the View.

This approach might appear excessively strict, but has some advantages: the communication network is artificially constrained, making it easier to evaluate and debug. All action happens in the Adapter, and the View can be created from off-the-shelf widgets without any Model-specific variation which make him more generic.

MVA is an implementation of the Mediator pattern. Controllers are generally referred as Adapters or Mediators. The Model and the View do not hold references to each other, they do not exchange data nor interact directly.



Create the Adapter - add a new Kotlin class and name it ItemAdpter it should inherit from the **RecyclerView.Adapter** but first let's create our View Holder. A View Holder is like its name suggest a class the holds references to all of our cell's views and given the data class it will bind the data to the views. Our View Holder should inherit from the **RecyclerView.ViewHolder** receive the binding object in its constructor and pass the root to his parent. Then given a data object it will bind the views to their data. So our View Holder should look like this:

```

class ItemAdapter {

    class ItemViewHolder(private val binding: ItemLayoutBinding)
        : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Item) {
            binding.itemTitle.text = item.title
            binding.itemDescription.text = item.description
        }
    }
}

```

To complete our class definition, define a primary constructor receiving the Items and inherit from the RecyclerView.Adapter (use our ItemViewHolder for the generic view holder). To get rid of the not implementing compilation error press ctrl+I and implement the three abstract functions. Our Adapter should look like this :


```
class ItemAdapter(val items:List<Item>) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
        TODO( reason: "Not yet implemented")
    }

    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
        TODO( reason: "Not yet implemented")
    }

    override fun getItemCount(): Int {
        TODO( reason: "Not yet implemented")
    }
}
```

Let's explain a bit about how the adapter is working:

It responds to the RecyclerView requests. First of all the RecyclerView gets the amount of items by calling the **getItemCount()** function and if the amount is bigger than zero, it asks for them one by one from its Adapter. Now, notice there are two functions for this: the create function and the bind function. As I explained before the first cells displayed on the screen (+one more) needs to be created from scratch, so for them the recycler calls both **onCreateViewHolder** supplying himself as the parent and a type (like we said this is used in case where one recycler cells has more then one layout file), the function returns the newly created ViewHolder, and with it, the recycler calls the **onBindViewHolder** passing the already created empty view holder and the relevant position, and this function use view holder's bind function with the specific Item at the requested position and return cell with the relevant data so he can Add it to the list and show the user. But as we said before when the user start scrolling the scrolled out cell moves to the recycle bin and then the recycler doesn't have to call the "expensive" **onCreateViewHolder** but only the "cheap" and fast **onBindViewHolder** function. So it's the recycler choice when to create the cell or just bind the data according to what it has in its recycle bin.

So out full Adapter code should look like this:

```
class ItemAdapter(private val items:List<Item>) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        ItemViewHolder(ItemLayoutBinding.inflate(LayoutInflater.from(parent.context),parent, attachToParent: false))

    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) = holder.bind(items[position])

    override fun getItemCount() = items.size

    class ItemViewHolder(private val binding: ItemLayoutBinding)
        : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Item) {
            binding.itemTitle.text = item.title
            binding.itemDescription.text = item.description
        }
    }
}
```

Later we will get back to it and update the image as well as handling view events, but for now it's enough.

Connecting the Recycler to the Adapter and setting the Layout Manager

What we need to do now is to connect between the Recycler and the Adapter but before that we need to set a Layout Manager to the Recycler View.

A Layout manger decides how the cell will be organized. We have three options:

1. **LinearLayoutManager** - organizes the cells one after the other like a scrolling list, it can be either horizontal or vertical.
2. **GridLayoutManager** - organizes the cells in a grid or a table where we must supply number of columns.
3. **StaggeredGridLayoutManager** - is the same as before but each square in the grid can have a different height (like the notes app)

We will use the Linear Manager. So in the onCreateView or onViewCreated in All Items Fragment we set adapter and the layout manager of the Recycler view we have added before to the xml file. In the **onViewCreated** after setting the layout manager pass the List of items from the ItemManger object to the Adapter's constructor and set him is our recycler view's adapter. So our almost finished code should look like this:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.recycler.layoutManager = LinearLayoutManager(requireContext())
    binding.recycler.adapter = ItemAdapter(ItemsManger.items)
}
```

Before we run the app and see how the magic happens we first need to add items to the list. So add the new item to the ItemManger object and remove the bundle from the navigate action. Your code should look like this:

```
binding.finishBtn.setOnClickListener { it: View!

    val item = Item(binding.itemNameEt.text.toString(),
        binding.itemDescriptionEt.text.toString(), photo: null)
    ItemsManger.add(item)

    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment)
}
```

Run your app and test your recycler. It's working very nicely but be aware of the fact that the list not saved to the file system.

Receiving events from Recycler

First of all, unlike other Views the Recycler doesn't have any interface throughout which it can send us user events.

Having said that, If all you want is dragging and swiping you have a pre-made Helper you can attach recycler like this:

```
binding.recycler.layoutManager = LinearLayoutManager(requireContext())
binding.recycler.adapter = ItemAdapter(ItemsManger.items)

ItemTouchHelper(object : ItemTouchHelper.Callback() {

    override fun getMovementFlags(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder
    ) = makeFlag(ACTION_STATE_SWIPE, directions: LEFT or RIGHT)
        //or makeFlag(ACTION_STATE_DRAG, UP or DOWN or LEFT or RIGHT)

    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean {
        TODO(reason: "Not yet implemented")
    }

    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        ItemsManger.remove(viewHolder.adapterPosition)
        binding.recycler.adapter!!.notifyItemRemoved(viewHolder.adapterPosition)
    }
}).attachToRecyclerView(binding.recycler)
```

The **getMovementFlags()** return the available gestures (swiping and / or dragging) and to and from which direction.

The **onMove()** function is called upon dragging event

And the **onSwiped()** upon swiping. Since it's all we allow here, we implement only it and remove the item from the data. But the adapter which also has a reference to this list should know that its data source has changed and it needs to notify the recycler to get updated and read the data again. There are few functions throughout which the adapter can cause the recycler to get updated one is **notifyDataSetChanged()** which causes the recycler to read all of the adapter data all over again but if have a more specific change we can use the **notifyItemRemoved/Inserted/Updated** and pass them the exact index of the update. Besides the fact that it's more efficient it's also done with Animation.

Receiving custom events from our RecyclerView

This is a bit more tricky. Because when we want get a custom event we need to attach a listener to the Views, but the only one who has access to these views is the adapter and if we write the event handing code in the adapter we decide on one implementation for all and loose our ability to be generic and let the class using the Adapter decide of its own event handling. Like here where the fragment is implementing the **onSwiped()**.

We want to create exactly this. We want to separate the event from the event handling, and let another class decide for itself on how to respond to the event

we are reporting about.

So first decide on the events you want to report about and the info you want to send with it and create your interface inside the Adapter:

```
class ItemAdapter(private val items:List<Item>){  
  
    interface ItemListener {  
        fun onItemClick(index: Int)  
        fun onItemLongClick(index: Int)  
    }  
}
```

Make the Adapter's constructor to receive an instance of that Listener (as well as the list) and make the View Holder class (the one receiving this actual events) to be an **inner** class so it can access this callback and invoke its functions (the functions that will be later implemented in the fragment for example).

```
class ItemAdapter(private val items:List<Item>, private val callback : ItemListener)  
    : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {
```

Our View holder should register for these view events and call the event callback function with the relevant info:

```

inner class ItemViewHolder(private val binding: ItemLayoutBinding)
: RecyclerView.ViewHolder(binding.root),
View.OnClickListener, View.OnLongClickListener {

    init {
        binding.root.setOnClickListener(this)
        binding.root.setOnLongClickListener(this)
    }

    override fun onClick(p0: View?) {
        callback.onItemClicked(adapterPosition)
    }

    override fun onLongClick(p0: View?): Boolean {
        callback.onItemLongClick(adapterPosition)
        return true
    }

    fun bind(item: Item) {
        binding.itemTitle.text = item.title
        binding.itemDescription.text = item.description
    }
}

```

In in the Fragment just implement this functions when you create the Adapter and show a Toast message:

```

binding.recycler.layoutManager = LinearLayoutManager(requireContext())
binding.recycler.adapter = ItemAdapter(ItemsManger.items, object : ItemAdapter.ItemListener {
    override fun onItemClicked(index: Int) {
        Toast.makeText(requireContext(), text: "${ItemsManger.items[index]}", Toast.LENGTH_SHORT).show()
    }

    override fun onItemLongClick(index: Int) {
        TODO(reason: "Not yet implemented")
    }
})

```

Last step - add the photo from the gallery

What we want to do is for the user to pick an image from the gallery (later on we will add also the camera option). This is part of the Start Activity For Result API which creates a new Activity and returns its result.

Let's go back the Runtime Permissions, the same mechanism is implemented here. In fact the runtime permission was just a single use case of the entire Launchers API mechanism. This mechanism is discussed in details in its own module. In short the idea behind it is to register a launcher in the activity or

fragment creation and when we need to, launch it - the idea is to make the call and the result not dependent in each other.

Here we use a different contract then the request permission contract, the contract will be the **OpenDocument()** contract in which the launcher receives an array of strings representing the mime types of the files we want to show to the user to choose one from. The result is the Uri of the specific file chosen by the user. Because we launch another component which reads the file storage and display it to the user for him a choose from, we don't need to ask for the reading permission ourself. Instead the activity that actually read the storage should ask for the permission.

So add the launcher creation to the AddItemFragment and in the callback that receives the URI of the photo chosen, set this photo in the image view and add it to the Item instance.

Be aware of the fact that for security reasons this Uri is temporary it is valid until our activity session will end(until **onDestroy()**). Because we need to save it in the file system later on, we need to ask for the OS to make the Uri persistent. This is done through the Content Resolver component that will be discussed later on as well as the actual saving of the item in the local DB.

Our Launcher definition will look like this:

```
private var imageUri : Uri? = null

val pickItemLauncher: ActivityResultLauncher<Array<String>> =
    registerForActivityResult(ActivityResultContracts.OpenDocument()) { it: Uri!
        binding.imageView.setImageURI(it)
        requireActivity().contentResolver.takePersistableUriPermission(it, Intent.FLAG_GRANT_READ_URI_PERMISSION)
        imageUri = it
    }
```

Replace the phot null value with the imageUri in the Item constructor call

```
val item = Item(binding.itemNameEt.text.toString(),
    binding.itemDescriptionEt.text.toString(), imageUri.toString())
ItemsManger.add(item)
```

In the Pick Photo button click launch you launcher and give the "image/*" mime type which means images of all types

```
binding.picButton.setOnClickListener { it: View!
    pickItemLauncher.launch(arrayOf("image/*"))
}
```

Now we see the photo in the ImageView but not yet in the recycler. To achieve this we need to go back to our bind function of the View Holder and use the external Open Source Glide library to read the image from the Uri stored in The

item class into the image view of the cell. The reason we use Glide is besides of its incredible images caching and auto resizing that greatly improves performance it is also doing all of its IO work on a background thread automatically and update the UI on the main thread - we don't need to worry about it - it is also done very efficiently.

So add the Glide dependency to the App grade file and sync your project (you can find the latest in the [Glide GitHub](#))

```
implementation 'com.github.bumptech.glide:glide:4.12.0'  
annotationProcessor 'com.github.bumptech.glide:compiler:4.12.0'
```

And in the bind function use it to load the image and make it nice and round into the image view:

```
fun bind(item: Item) {  
    binding.itemTitle.text = item.title  
    binding.itemDescription.text = item.description  
    Glide.with(binding.root).load(item.photo).circleCrop().into(binding.itemImage)  
}
```

That's it our project if finished for now. Later on we will move our data to the View Model, notify about changes in it with the Live Data and make it persistent with ROOM database.

APPENDIX - Parcelable and Serializable

If we need to pass the object from the Adding Fragment to the All Items Fragment we must put it in the bundle and sends it with the navigation. Please note that we can't put an object reference in the bundle since it is generally used to pass data between components(activities for example) and sending object references between Android components is not possible because by changing the process, the object references won't be in the new process, so we must make our objects Parcelable or Serializable. It means turning them into streams of bytes and put it in the the Bundle. Serializable is simpler since it doesn't require implementing any methods but with more overhead since all the work is done at runtime and reflection in general cost more in terms of efficiency so we would rather use Parcelable because its is built for that exact purpose and is highly optimized for IPC (Inter Process Communication). But there are allot of functions to add so we rather use the kotlin-parcelize plugin in the app Gradle file:

```
ent.kt x build.gradle (:app) x ac
n use the Project Structure dialog to v
plugins {
    id 'com.android.applicat
    id 'kotlin-android'
    id 'kotlin-parcelize'
}
```

This plugin along with the @Parcelize Annotation in the class definition will cause the compile to generate all the Parcel functions for us

```
@Parcelize
data class Item(val title:String, val description: String, val photo: String?) : Parcelable
```

Another option without any plugin or annotations is to implement the Serializable interface which all of his functions are added in runtime

```
data class Item(val title:String, val description: String, val photo: String?) : Serializable
```

And that's it, now we can send it in the Bundle and add it to the list. But the adapter which also reference this list should know that it's data source has changed and it needs to notify the recycler to get updated and read from it the most relevant data. There are few functions through which the adapter can cause the recycler to get adapted one is **notifyDataSetChanged()** which causes the recycler to read all of the adapter data all over again but if we have a more specific change we can use the **notifyItemRemoved/Inserted/Updated** and pass them the exact index to refresh the view.

So our code for sending the Item in the Add Item Fragment should look like this:

```
binding.finishBtn.setOnClickListener { it: View!
    val bundle = bundleOf( ...pairs: "item" to Item(binding.itemNameEt.text.toString(),
                                                binding.itemDescriptionEt.text.toString(),
                                                photo: null))
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment,bundle)
}
```