Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

UI Navigation

Federico Montori
federico.montori2@unibo.it

Table of Contents

- Fragments
  - Fragment Transactions
- Preferences
- Dialogs
- App Bar
  - Menu
  - Drawers
- Navigation Framework

Fragments

Fragment → A portion of the user interface in an Activity.

Basically, a Fragment is a modular section of an Activity (a FragmentActivity).

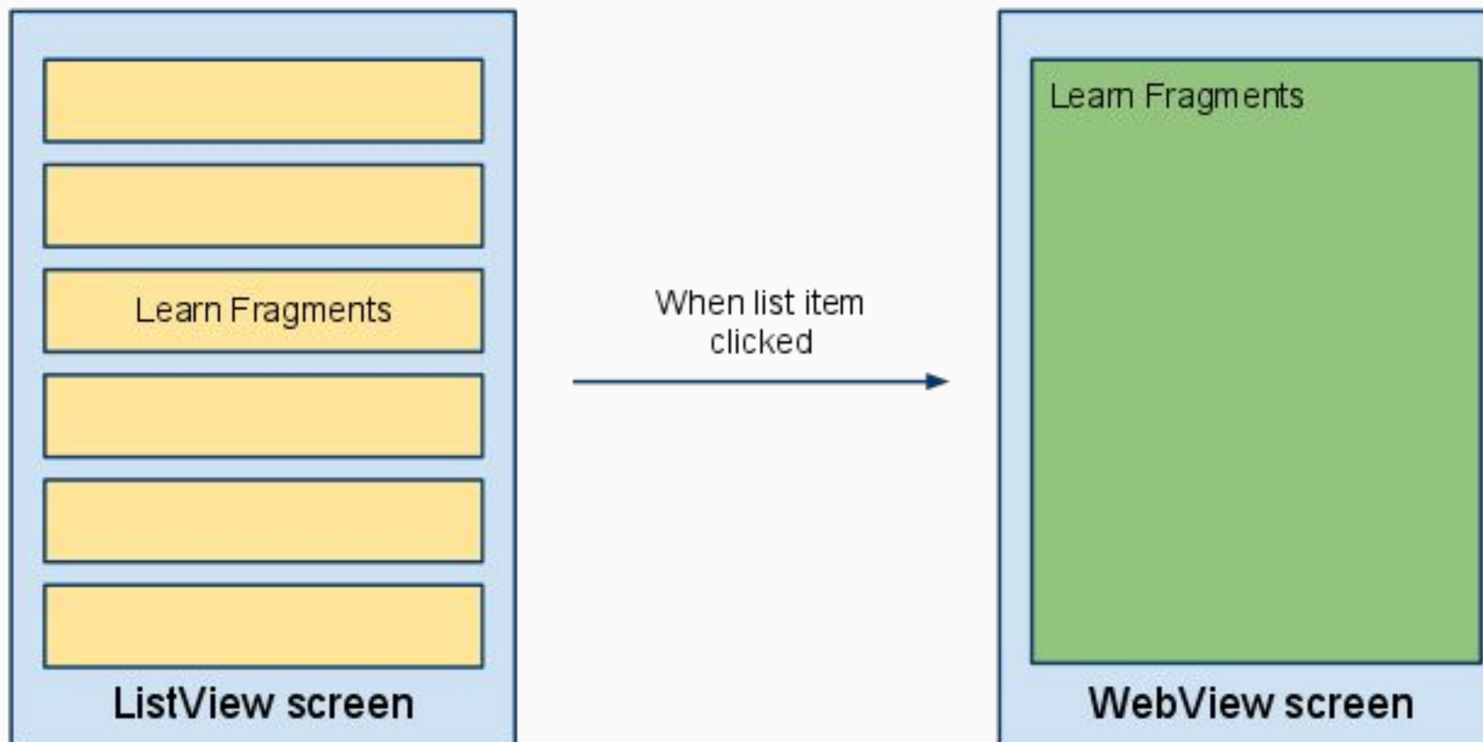
→Introduced in Android 3.0 (API Level 11)

DESIGN PHILOSOPHY:

- Structure an Activity as a *collection* of Fragments.
- *Reuse* a Fragment on different Activities …

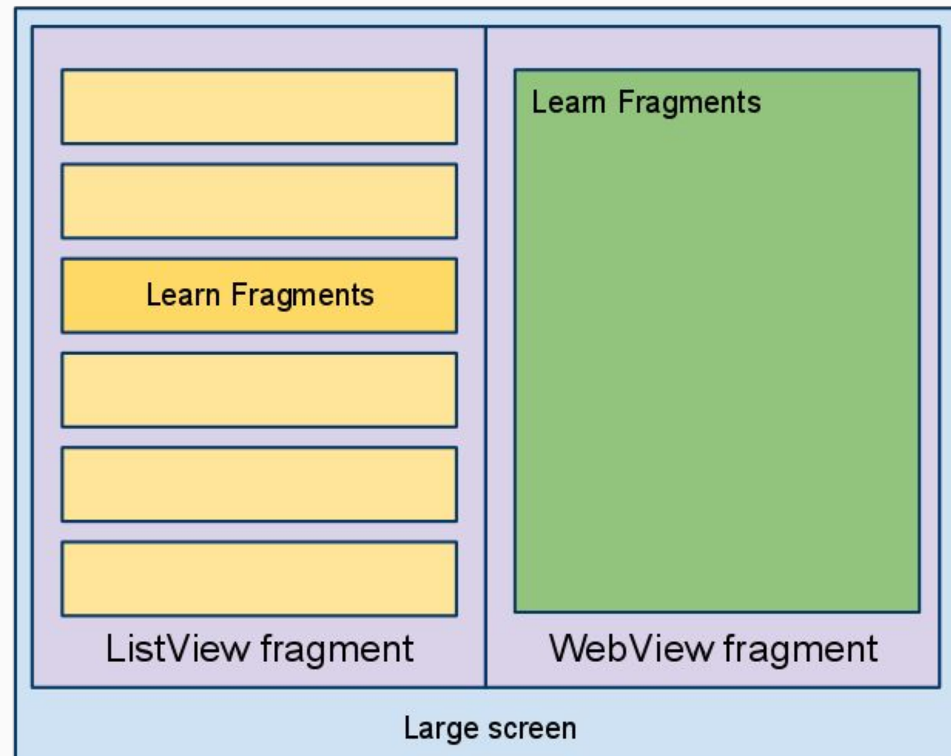
Fragments

EXAMPLE: Structuring an Application using multiple Activities.



Fragments

EXAMPLE: Structuring an Application using 1 Activity and 2 Fragments.



Fragments

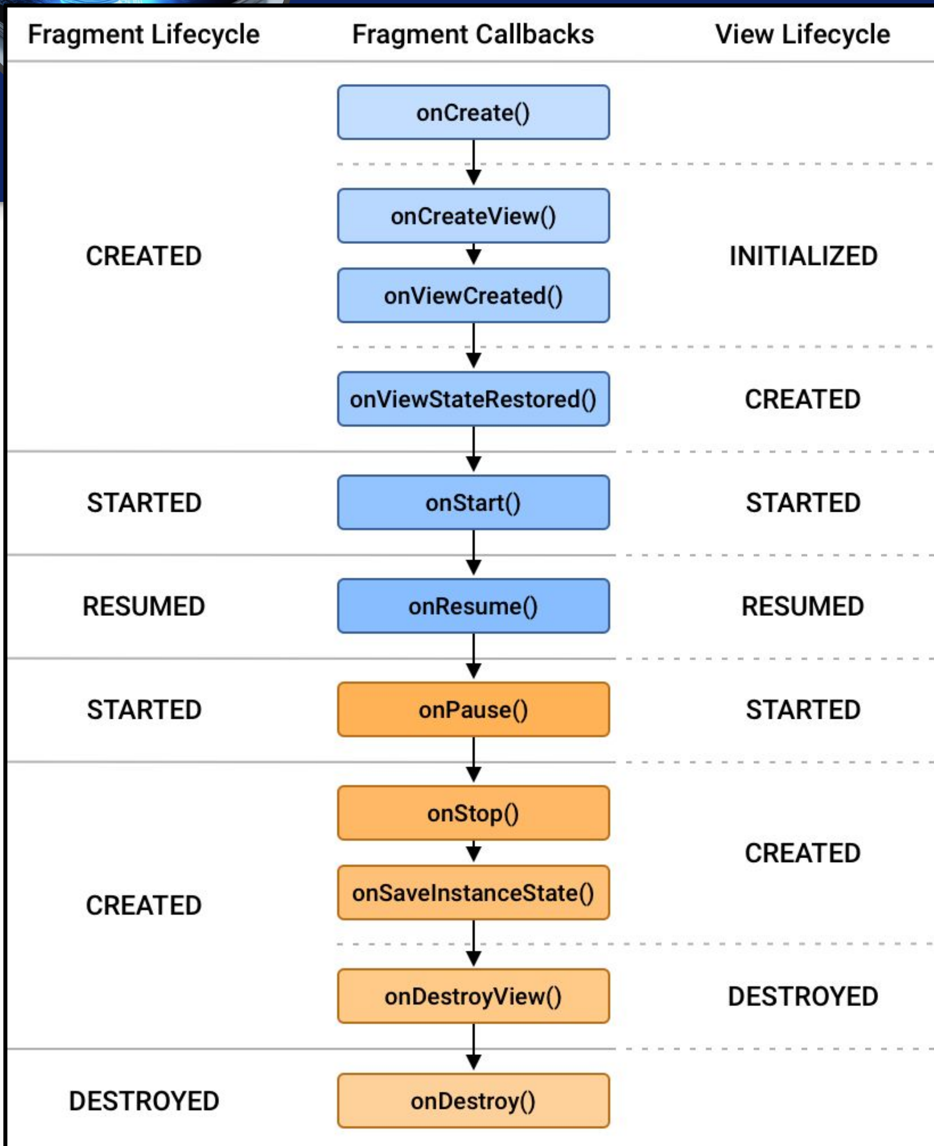
To define a new Fragment → create a subclass of Fragment.

```
class BlankFragment : Fragment() { ... }
```

Properties:

- Has its own lifecycle (partially connected with the Activity lifecycle)
- Has its own layout (or may have)
- Can receive its own input events
- Can be added or removed while the Activity is running.
- Cannot run by itself (always hosted by an Activity)
- Cannot receive Intents!

Fragments

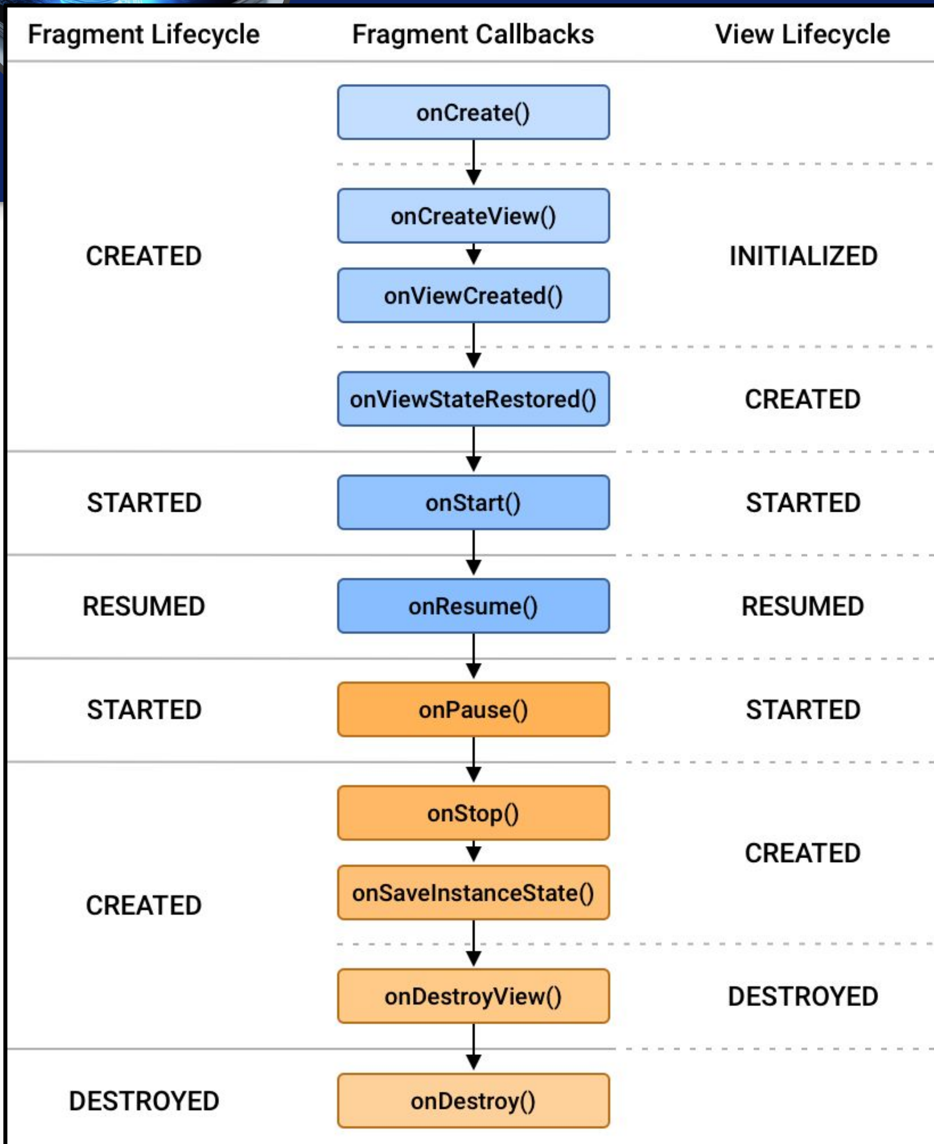


Several callback methods to handle various stages of a Fragment lifecycle.

onCreate() → called when creating the Fragment (elements retained when stopped).

onPause() → called when the user is leaving the Fragment (commit changes in need of persistence).

Fragments

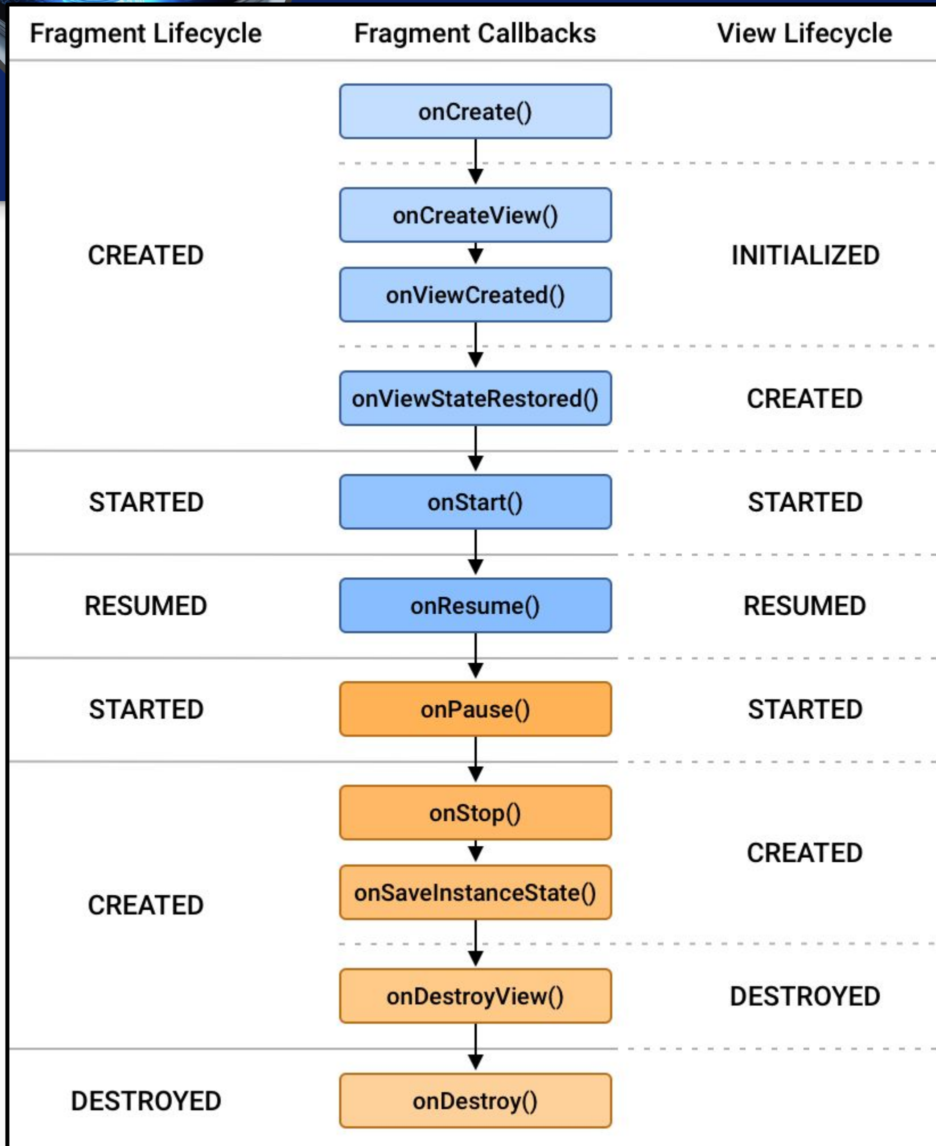


The lifecycle of the Activity in which the Fragment lives directly affects the lifecycle of the Fragment.

- onPause (Activity) → onPause (Fragment)
- onStart (Activity) → onStart (Fragment)
- onDestroy (Activity) → onDestroy (Fragment)

Fragments have also extra lifecycle callbacks to enable runtime creation/destruction.

Fragments



Several callback methods to handle various stages of a Fragment lifecycle.

onCreateView() → called when it is time for the Fragment to draw the user interface the first time (or coming back from the backstack).

Good to set the properties in **onViewCreated**().

Fragments

onCreateView() → must return the View associated to the UI of the Fragment

Use a LayoutInflater

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle? ): View? {
        return inflater.inflate(R.layout.fragment_blank, container, false)
}
```

This is pretty much of a boilerplate… more recently you can do the same with:

```
class BlankFragment : Fragment(R.layout.fragment_blank)
```

Fragments

Add it to the layout of your Activity like so:

```
<fragment android:name="com.example.BlankFragment"
    android:id="@+id/blankfragment"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    />
```

This is going to be a **Static Fragment**, i.e. it cannot be replaced or moved and it is very basic.

See **Transactions** for a better way to do this…

Fragments

Once specified, here's what the system does:

- Assigns the layout to the Activity in the usual way
- Creates all the fragments by instantiating the classes and calling the onCreate() method.
- It calls the onCreateView() so, though the inflater, the fragment tells:
  - what is the fragment content in terms of view (par 1)
  - and where to put it (usually the container passed to the function) (par 2)

You can always do this programmatically instead

Fragments

FragmentManager → a support API element that handles the Fragments' lifecycle and scheduling:

The FragmentManager manages the Fragments associated within the context.

Level	Instance of		Fragment Manager
Host Activity	FragmentActivity	supportFragmentManager	FragmentManager
Host Fragment	Fragment	parentFragmentManager childFragmentManager	FragmentManager
Child Fragment(s)	Fragment	parentFragmentManager	

Each FragmentManager manages child fragments of the host

Fragments

A Fragment can get a reference to the Activity:

```
getActivity()
```

An Activity can get a reference to the Fragment:

```
supportFragmentManager.findFragmentById(R.id.blankfragment)
```

Before a Fragment enters the lifecycle, it calls its **onAttach**() method right when it gets passed to the FragmentManager.
The dual is **onDetach**().

Fragments

If you need the activity to react strictly to the Fragment events:

```
override fun onAttach(context: Context) {
    super.onAttach(context)
    try {
        val myListener = context as MyListener
    } catch (e: ClassCastException) {
        /* The calling activity is not implementing the MyListener interface */
    }
}
```

Fragment has to expose an interface that the activity must implement and the Fragment checks it in the onAttach() (activity is passed here)...

Fragments

If you need the activity to react strictly to the Fragment events:

```
override fun onAttach(context: Context) {
    super.onAttach(context)
    try {
        val myListener = context as MyListener
    } catch (e: ClassCastException) {
        /* The calling activity is not implementing the MyListener interface */
    }
}
```

Fragment has to expose an interface that the activity must implement and the Fragment checks it in the onAttach() (activity is passed here)…

Fragment Transactions

- Fragments can be added/removed/replaced while the Activity is running …
- Each set of changes to the Activity is called a Transaction.
- Transaction can be saved in order to allow a user to navigate backward among Fragments when he clicks on the “Back” button.

For these Dynamic Fragments you should specify a **FragmentContainerView**

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/blankfragment"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    />
```

Fragment Transactions

Perform a transaction on the FragmentContainerView:

```
val transaction: FragmentTransaction = supportFragmentManager.beginTransaction()
transaction.run {
    setReorderingAllowed(true)
    replace(R.id.blankfragment, BlankFragment::class.java, null)
    addToBackStack("MyLabel")
    commit()
}
```

This will inject BlankFragment into the container, replacing whatever was there…

```
/* The fragment-ktx module provides a commit block that automatically calls
beginTransaction and commit for you. */
supportFragmentManager.commit { … }
```

Fragment Transactions

A Transaction is not performed till the commit …

- If **addToBackStack**() is not invoked the old Fragment is destroyed and it is not possible to navigate back.
- If **addToBackStack**() is invoked the old Fragment is stopped and it is possible to resume it when the user navigates back.
- **popBackStack**() simulates a Back from the user.

Both Fragments and Activities make use of a backstack, however:

- The backstack of the activities is kept by the system, whereas the backstack of the fragments is kept by the host activity.
- Saving a fragment to the backstack has to be explicitly requested.

Fragment Transactions

With **FragmentContainerView** new Fragments can be replaced easily.

- Layout of Fragments have always to be within a **FrameLayout** (not necessarily true for <**fragment**>).
- If FragmentContainerView has an *android:name* or a class then it triggers a Fragment Transaction when the Activity starts up.

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/blankfragment"
    android:name="com.example.navigationexample.BlankFragment"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    />
```

Shared Preferences

- SharedPreferences are a convenient way to store configuration parameters on the disk
- Structured with a key-value mode

Shared Preferences	
Preference **TAG**	Preference **VALUE**
Preference **TAG**	Preference **VALUE**
Preference **TAG**	Preference **VALUE**
Preference **TAG**	Preference **VALUE**

Shared Preferences

SharedPreferences could be either **private** or **public**

- Public means that other applications could potentially read such preferences
  - only until Android 7, now for that you must use a **ContentProvider**
- Private means that they could be restricted at
  - Application level
  - Activity level

We can also set a Preference screen, by using the Settings API from Jetpack.

Shared preferences are identified as a **Bundle** of data

Shared Preferences

From the activity…

```
getSharedPreferences("name of the bundle", Context.MODE_PRIVATE)
```

Get a reference to the preference file identified by the string parameter

```
getPreferences(Context.MODE_PRIVATE)
```

Get a reference to the default preference file associated to the calling activity

```
PreferenceManager.getDefaultSharedPreferences(this)
```

Get a reference to the default shared preferences, which are also used to create the **Preference Screen**

Shared Preferences

Get values

```
preferences.getString("key", "default value") // return default value if key does not exist
```

Edit the preferences

```
with (preferences.edit()) { // This will give back a SharedPreferences.Editor object
   putString("key", "new value")
   commit()
}
```

Alternatively you could also call apply() instead of commit, which writes the data to the disk asynchronously. Calling commit() stops your main thread.

Shared Preferences

You can interact with the default SharedPreferences through preferences screen.

- Starting with Android 10, android.preference is deprecated.
- Use Androidx Preference Library (or Settings API) instead.

add:

```
implementation("androidx.preference:preference-ktx:1.2.0")
```

- It comes with a built-in Material Design look and feel
- It uses the res/xml resource directory

Shared Preferences

- Use either basic Preference if you want no widget
- Use a facility if you want widgets
  - https://source.android.com/devices/tech/settings/settings-guidelines

res/xml/my_pref.xml

```xml
<PreferenceScreen
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <SwitchPreferenceCompat
    app:key="notifications"
    app:title="Enable message notifications"/>
  <Preference
    app:key="feedback"
    app:title="Send feedback"
    app:summary="Report technical issues or suggest new features"/>
 </PreferenceScreen>
```

Shared Preferences

To inflate the XML hierarchy on a screen, just extend the PreferenceFragment and display it as you would with any other Fragment.

- Remember, this works with the **DefaultSharedPreferences**

```
class MySettingsFragment : PreferenceFragmentCompat() {
   override fun onCreatePreferences(savedInstanceState: Bundle?, rootKey: String?) {
      setPreferencesFromResource(R.xml.my_pref, rootKey)
   }
}
```

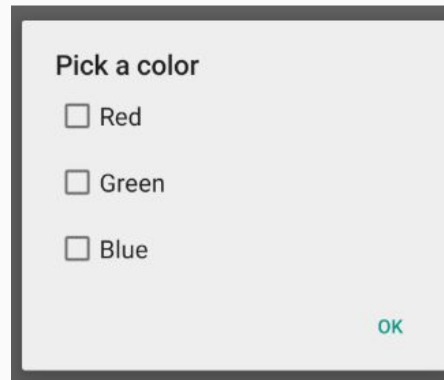
Dialog

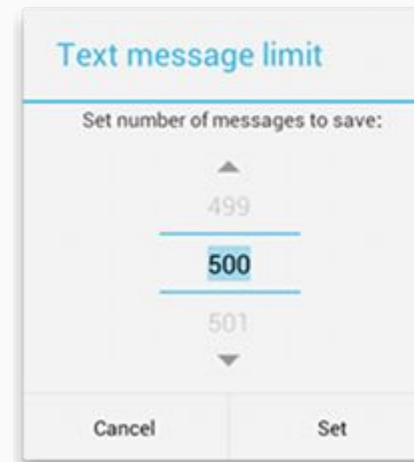
Used to interact with the user

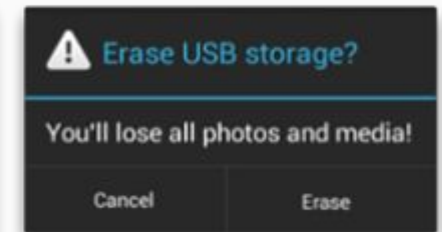
Little messages, easy answers

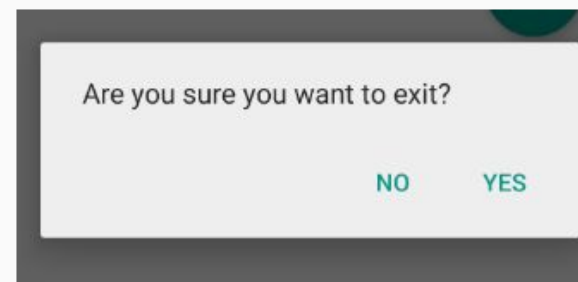
Different kinds:

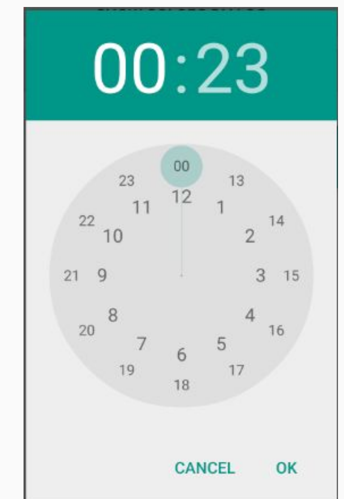
- AlertDialog
- DatePickerDialog
- TimePickerDialog
- **DialogFragment**
    - This is an actual Fragment











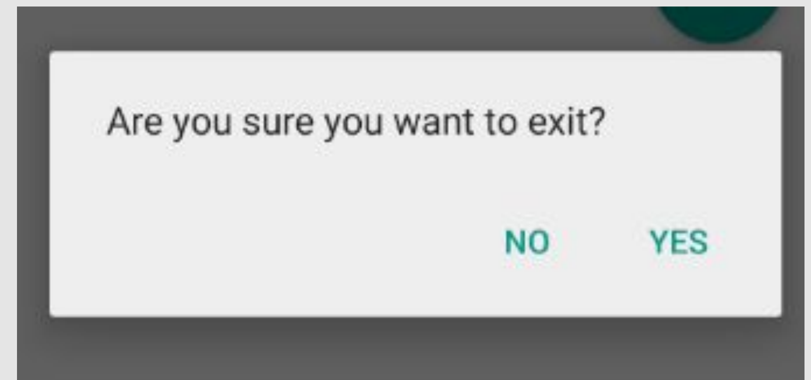
Dialog

```
val builder = AlertDialog.Builder(this)
builder.also {
   it
      .setMessage("Are you sure you want to exit?")
      .setCancelable(false)
      .setPositiveButton("Yes", { dialog, id -> finish() })
      .setNegativeButton("No", { dialog,id -> dialog.cancel() })
}

val alert: AlertDialog = builder.create()

alert.show()
```

cancelable through back?



Dialog

```
val items = arrayOf("Red", "Green", "Blue")
val builder = AlertDialog.Builder(this)
builder.also {
    it
        .setTitle("Pick a color")
        .setItems(items) { dialog, item ->
            Toast.makeText(this, items[item], Toast.LENGTH_LONG).show()
        }
}

val alert: AlertDialog = builder.create()

alert.show()
```

Dialog

For non-standard Dialogs you should implement a **FragmentDialog**

```
class PurchaseConfirmationDialogFragment : DialogFragment() {
   override fun onCreateDialog(savedInstanceState: Bundle?): Dialog =
         AlertDialog.Builder(requireContext())
            .setMessage(getString(R.string.order_confirmation))
            .setPositiveButton(getString(R.string.ok)) { _,_ -> }
            .create()
}
PurchaseConfirmationDialogFragment().show(childFragmentManager, "Header")
```

For a more customized behavior you can override **onCreateView**() and **onViewCreated**() as usual.

App Bar

*"In its most basic form, the **action bar** displays the title for the activity on one side and an overflow menu on the other. Even in this basic form, the app bar provides useful information to users and gives Android apps a consistent look and feel."*

```
<com.google.android.material.appbar.AppBarLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary" />
</com.google.android.material.appbar.AppBarLayout>
```

Use the **toolbar** tag and wrap it into an **AppBarLayout** to give it a bunch of other behaviors such as swipe events etc...

App Bar

The toolbar will float over your activity displaying access to the **Navigation Drawer** and the **Menu**.

Set an appropriate theme sticking to “**NoActionBar**” to prevent the system using the default action bar.

In your Activity:

```
setSupportActionBar(findViewById(R.id.my_toolbar))
```

If the action bar is set, we can do cool things such as changing the home image next to the app name…

```
supportActionBar?.apply {
    setDisplayHomeAsUpEnabled(true)
    setHomeAsUpIndicator(R.drawable.ic_launcher_foreground)
}
```

Menu

It appears whenever the user presses the **menu** button on the **app bar**

- Useful for giving different options without leaving the current Activity
- Don’t make menus too big, or they’ll cover entirely the Activity

The menu is declared in **XML**

- Place a file inside **res/menu/**
- Inflate the menu inside the Activity
- Useful if you want to create the same menu inside different activities

Menu

Create **res/menu/menu.xml**

We need:

- IDs of menu elements
- Title of each element
- Icon of each element

Inside the Activity, create onCreateOptionsMenu()

- Inflate the menu
- Add functionality to the buttons

Menu

Create **res/menu/menu.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/item1" android:title="First Option"></item>
  <item android:id="@+id/item2" android:title="Second Option">
    <menu>
      <item android:id="@+id/item3" android:title="Third Option"/>
      <item android:id="@+id/item4" android:title="Fourth Option"/>
    </menu>
  </item>
</menu>
```

Menu

Then in your Activity inflate the menu in its place:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.menu, menu)
    return super.onCreateOptionsMenu(menu)
}
```

This is is the procedure for the **Options Menu** belonging to the toolbar, but there are other kinds of menu (Contextual Menu, Popup Menu, …).

Menu

React from your activity to Menu click events:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.item1 -> {
            // Do your stuff
            true
        }

            [...]

        else -> super.onOptionsItemSelected(item)
    }
}
```

NavigationDrawer

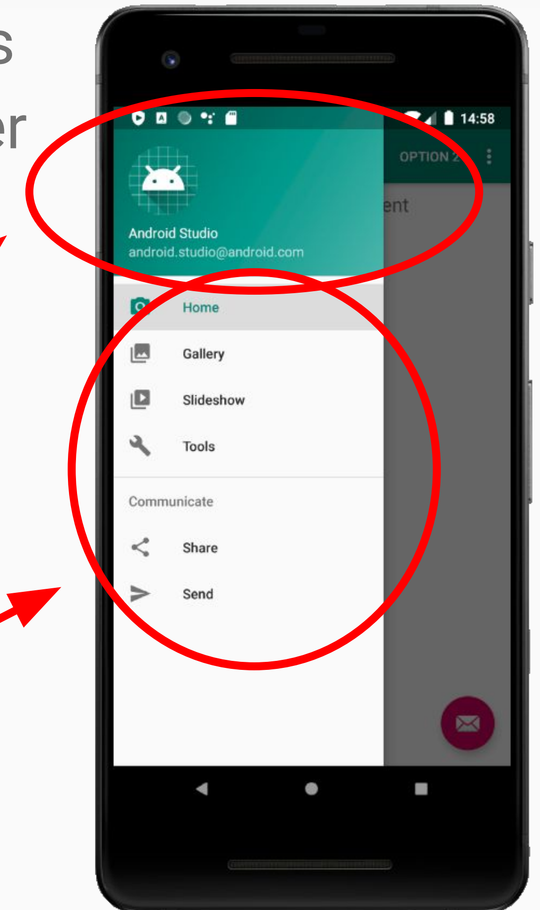
Novel navigation component, hidden when not in use, appears when swiping from the left or by clicking on the top-left drawer icon, if bound to an action bar.

res/layout/nav_header_main.xml

```
<LinearLayout ...>
   <ImageView ... /> <TextView ... /> <TextView ... />
</LinearLayout>
```

```
<menu>< group android:checkableBehavior="single">
                  <item ... />  <item ... />  <item ... />  <item ... />
            </group>
      <item android:title="Communicate">
            <menu> <item ... />  <item ... /> </menu>
      </item>
</menu>
```

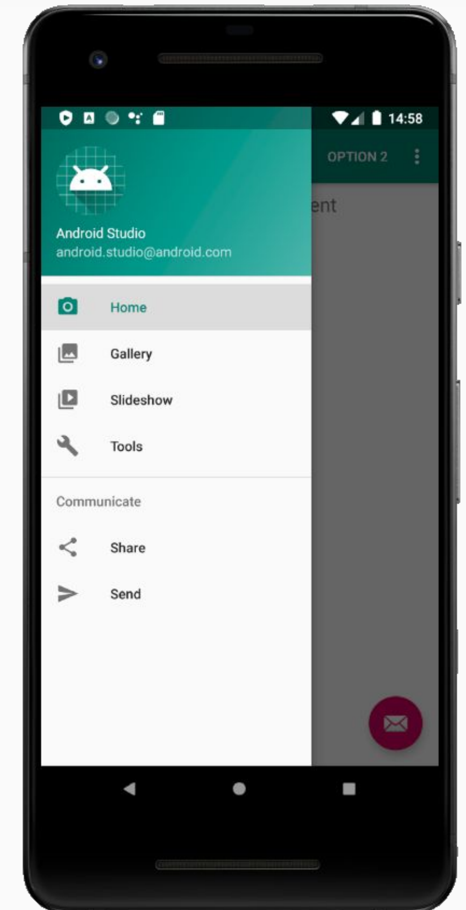
res/menu/activity_main_drawer.xml



NavigationDrawer

Novel navigation component, hidden when not in use, appears when swiping from the left or by clicking on the top-left drawer icon, if bound to an action bar.

```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:fitsSystemWindows="true"
    app:headerLayout="@layout/nav_header_main"
    app:menu="@menu/activity_main_drawer" />
```



NavigationDrawer

DrawerLayout should be added as root view inside the layout and has to contain:

- Layout when NavigationDrawer is hidden (YourMainLayout)
- Content of the navigation drawer (the NavigationView)

```
<androidx.drawerlayout.widget.DrawerLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:fitsSystemWindows="true"
   tools:openDrawer="start">

         <!--  you main layout ... -->
         <com.google.android.material.navigation.NavigationView ... />

</androidx.drawerlayout.widget.DrawerLayout>
```

NavigationDrawer

The Navigation Drawer responds to events as well…

```
val drawerLayout: DrawerLayout = findViewById(R.id.drawer_layout)
val navigationView: NavigationView = findViewById(R.id.nav_view)
navigationView.setNavigationItemSelectedListener { menuItem ->
        menuItem.setChecked(true)
        drawerLayout.closeDrawers()
        // do stuff on top of the menu item…
        true
}
```

To link properly a Navigation Drawer to the App Bar as we know it, let us use the **Navigation Framework**

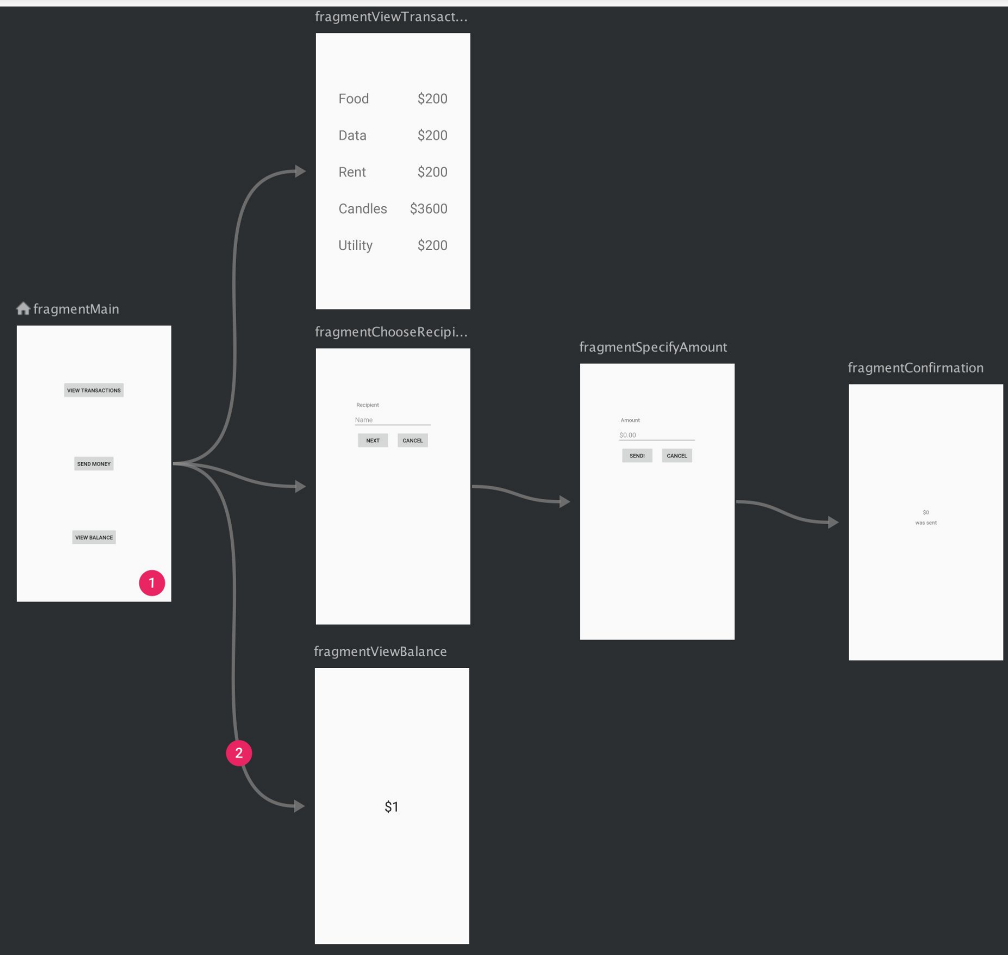
Navigation

Android Jetpack has launched the Android Navigation framework
https://developer.android.com/guide/navigation
Much easier way to handle navigation through:

- **NavHostFragment** (in practice you have 1 Activity with many fragments interleaving in the NavHostFragment as container).
- **NavigationController** (the central brain)
- **A Navigation Graph**

Remember: Navigation is sourced into a Nav host fragment: an empty container within which the navigation takes place. There may be an Activity change, although infrequent.

Navigation



The **Navigation Graph**:

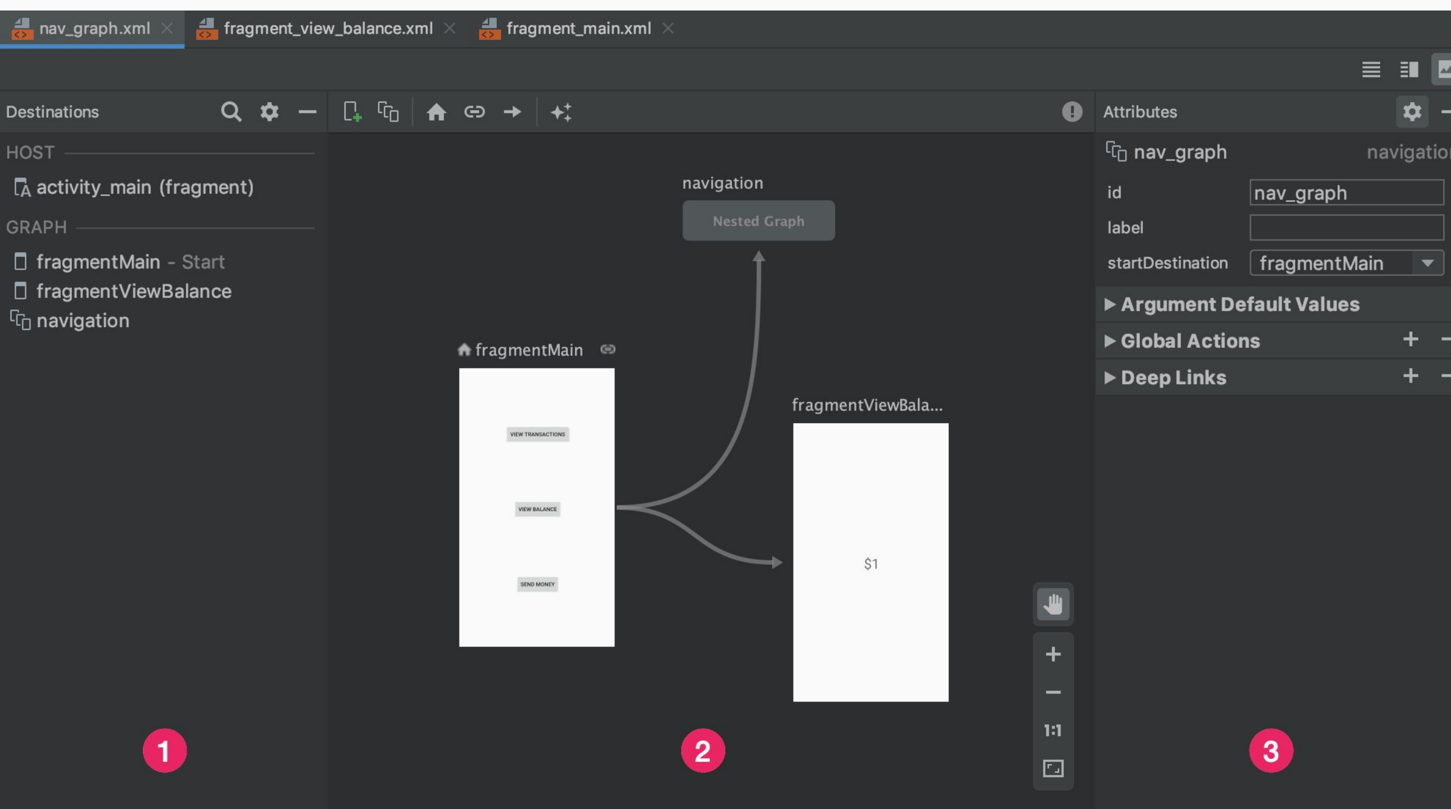
- An XML resource connecting **destinations** (fragments) through **actions** (events).
- The XML resource type is “**navigation**”.
- It must take place within a NavHostFragment (although destinations can also be activities).

Add the necessary dependencies…

```
implementation("androidx.navigation:navigation-fragment-ktx:2.7.7")
implementation("androidx.navigation:navigation-ui-ktx:2.7.7")
```

Navigation

You can edit the Navigation graph via the Navigation Editor.



1. **Destination panel**: you can see all your resources
2. **Graph Editor**: Contains a visual representation of your navigation graph. You can switch between Design view and the underlying XML representation in the Text view.
3. **Attributes**: Shows attributes for the currently-selected item in the navigation graph.

Navigation

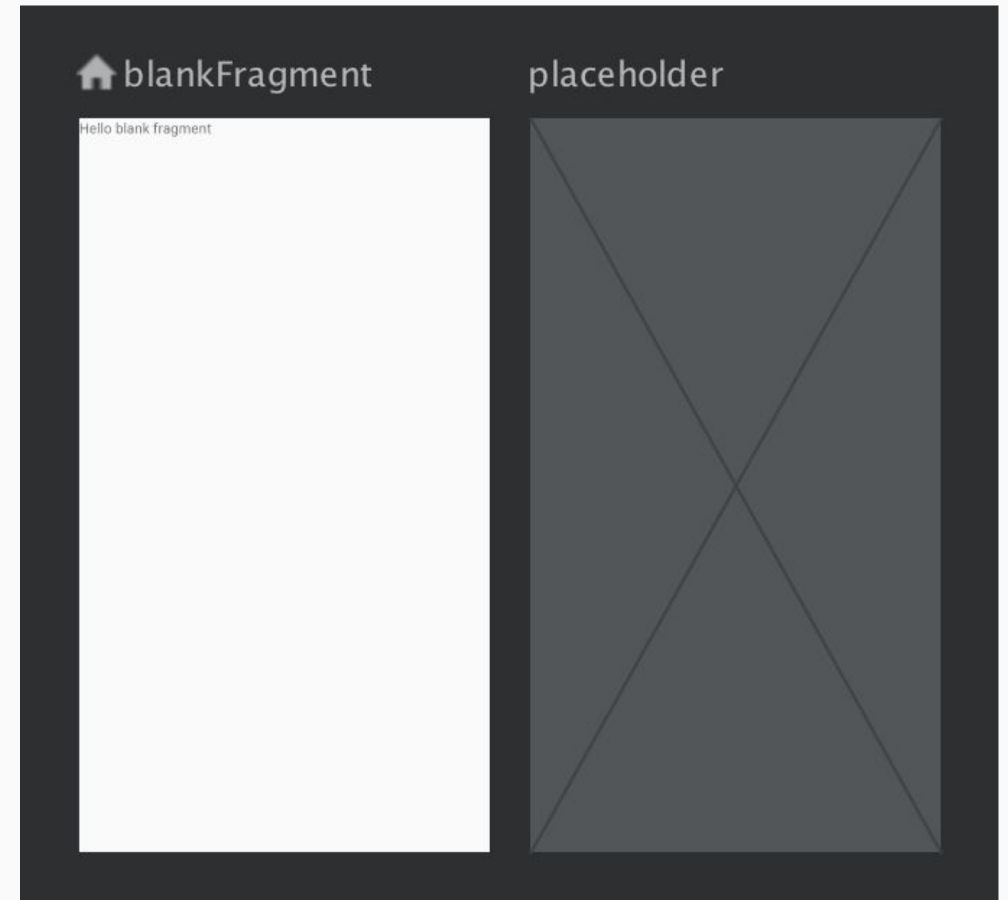
Instantiate the *Nav Host* in the activity where you want the Navigation to take place. This is implemented automatically by a class called **NavHostFragment**
Also specify to which navigation graph we are referring to by using the **navGraph** attribute.
defaultNavHost allows the fragment to intercept the back button.

```
<fragment  android:id="@+id/nav_host_fragment_content_main"
           android:name="androidx.navigation.fragment.NavHostFragment"
           android:layout_width="match_parent"
           android:layout_height="match_parent"
           app:defaultNavHost="true"
           app:navGraph="@navigation/mobile_navigation" />
```

Navigation

In creating a destination through the Editor you need to specify 4 different fields:

- The Type field indicates whether the destination is implemented as a fragment, activity, or other custom class in your source code.
- The Layout field contains the name of the destination's XML layout file.
- The ID field contains the ID of the destination which is used to refer to the destination in code.
- The Name dropdown shows the name of the class that is associated with the destination. You can click this dropdown to change the associated class to another destination type.

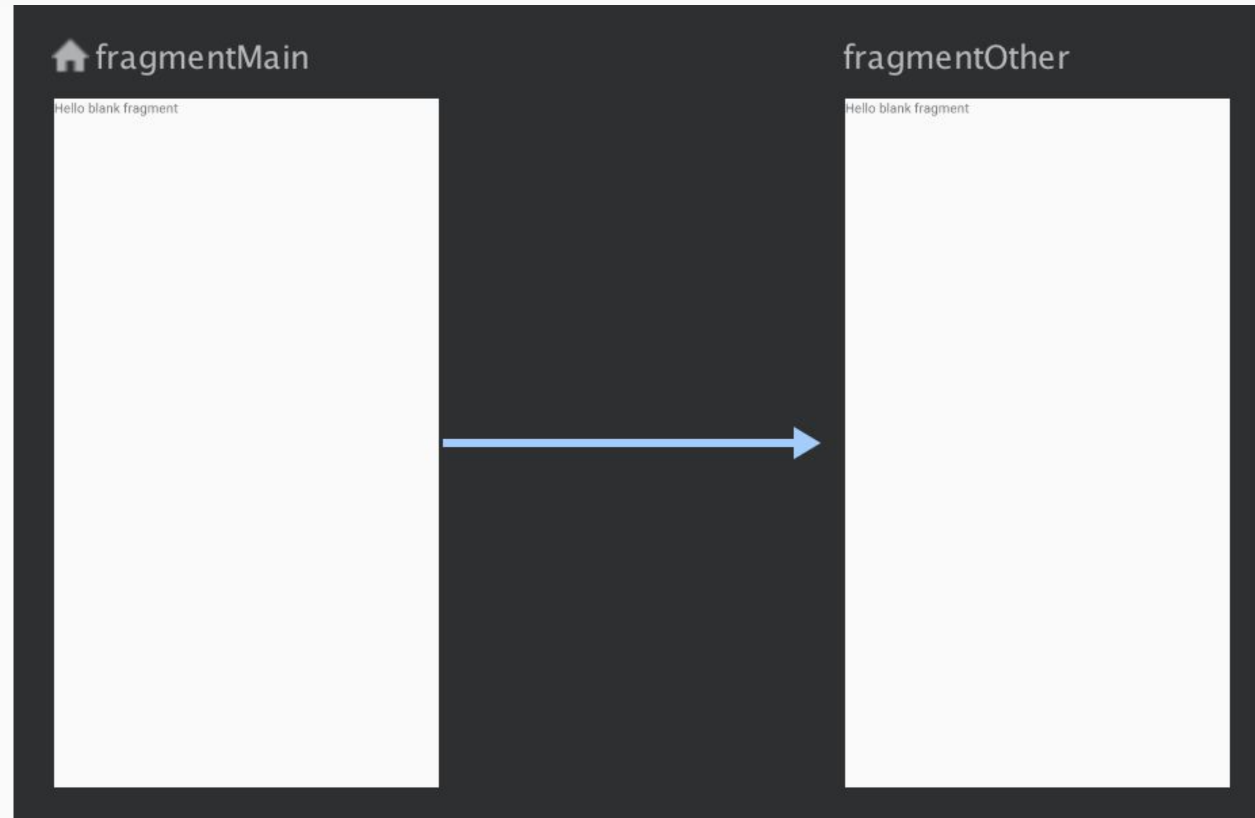


Navigation

In creating an action through the Editor you need to connect two destinations and specify 3 different fields:

- The Type field contains “Action”.
- The ID field contains the ID for the action.
- The Destination field contains the ID for the destination fragment or activity.

```
<action
    android:id="@+id/myaction"
    app:destination="@id/blankFragment2"
/>
```



Navigation

```xml
<navigation xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
xmlns:android="http://schemas.android.com/apk/res/android"
   app:startDestination="@id/blankFragment">
   <fragment
      android:id="@+id/blankFragment"
      android:name="com.example.cashdog.cashdog.BlankFragment"
      android:label="fragment_blank"
      tools:layout="@layout/fragment_blank" >
      <action
         android:id="@+id/myaction"
         app:destination="@id/blankFragment2" />
   </fragment>
   <fragment
      android:id="@+id/blankFragment2"
      android:name="com.example.cashdog.cashdog.BlankFragment2"
      android:label="fragment_blank_fragment2"
      tools:layout="@layout/fragment_blank_fragment2" />
</navigation>
```

Here’s how it will look like in the end…

In order to perform any action we need to retrieve the NavHostFragment:

```kotlin
val navHostFragment: NavHostFragment =
        supportFragmentManager
                .findFragmentById(
                R.id.nav_host_fragment_content_main
                ) as NavHostFragment
```

Navigation

Simply navigate by declaring the action:

```
navHostFragment.navController.navigate(R.id.myaction)
```

Or use **SafeArgs**, which ensure type safety

➔ Follow this to add the classpath https://developer.android.com/guide/navigation/use-graph/pass-data#kts

- Once enabled, it creates a class for each origin destination ensuring type safety when performing an action. The class is called {name_of_origin} + “Directions”
- Such class has a method for each of the actions that returns a NavDirection object to be passed to the navigate function.

Considering the previous XML:

```
val action: NavDirections = BlankFragmentDirections.myaction()
navHostFragment.navController.navigate(action)
```

Navigation

Finally, link the top-level destinations to the Navigation Drawer through the App Bar

```
val navController = findNavController(R.id.nav_host_fragment_content_main)
val drawerLayout: DrawerLayout = findViewById(R.id.drawer_layout)
val navView: NavigationView = findViewById(R.id.nav_view)

/* Populate the NavigationDrawer (elements in the navigation drawer menu must have the
same id as the top-level destinations in the navigation graph */
val appBarConfiguration = AppBarConfiguration(
    setOf( R.id.nav_home, R.id.nav_gallery, R.id.nav_slideshow ), drawerLayout
) // This also replaces the UP icon in the app bar with the "hamburger"
setupActionBarWithNavController(navController, appBarConfiguration)

/* This redirects the setNavigationItemSelectedListener to the navigation actions */
navView.setupWithNavController(navController)
```

Navigation

Navigation keeps a backstack of all the transactions and overrides the usage of the back button to navigate back the backstack.
It also sets a up button on the toolbar that does exactly the same thing as back, but it never exits the app (it is replaced by e.g. the navigation “hamburger” icon).
It creates a fake backstack if we deep link to a certain screen.

This behavior is handled through the following:

```
override fun onSupportNavigateUp(): Boolean {
    val navController = findNavController(R.id.nav_host_fragment_content_main)
    return navController.navigateUp(appBarConfiguration) || super.onSupportNavigateUp()
}
```

Questions?

federico.montori2@unibo.it