

AD340 Mobile Application Development

...

Week 05

Outline

- Image Resources
- Navigation within an app
- Custom navigation behavior
- Navigation UI
- Activity Lifecycle
- Logging
- Framework lifecycle
- Lifecycle-aware components
- Tasks and back stack

Last Week

Data Binding

RecyclerView

Multiple activities and intents

App Bar, Navigation drawer, and Menus

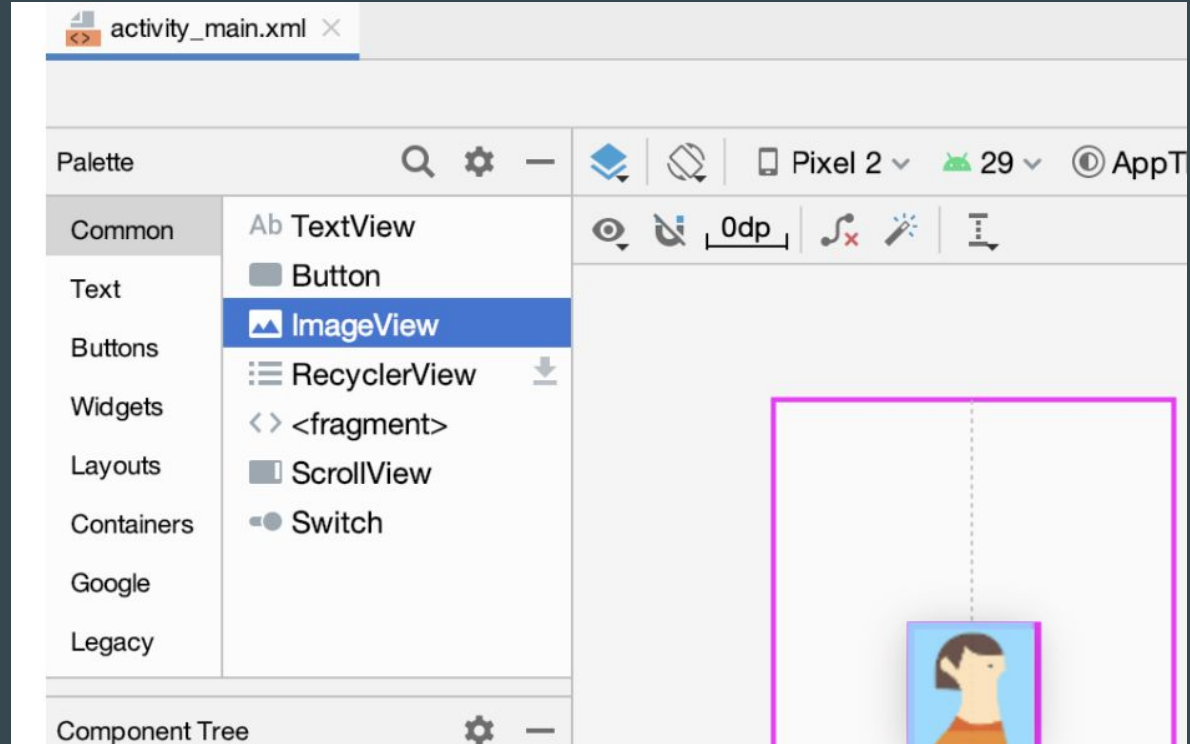
Fragments

Questions

Image Resources

ImageView onto the Design view

First you want to drag an ImageView onto the Design view



Add Image resources

click on View > Tool Windows > Resource Manager in the menus

Or

click on the Resource Manager tab to the left of the Project window

Click the + below Resource Manager, and select Import Drawables

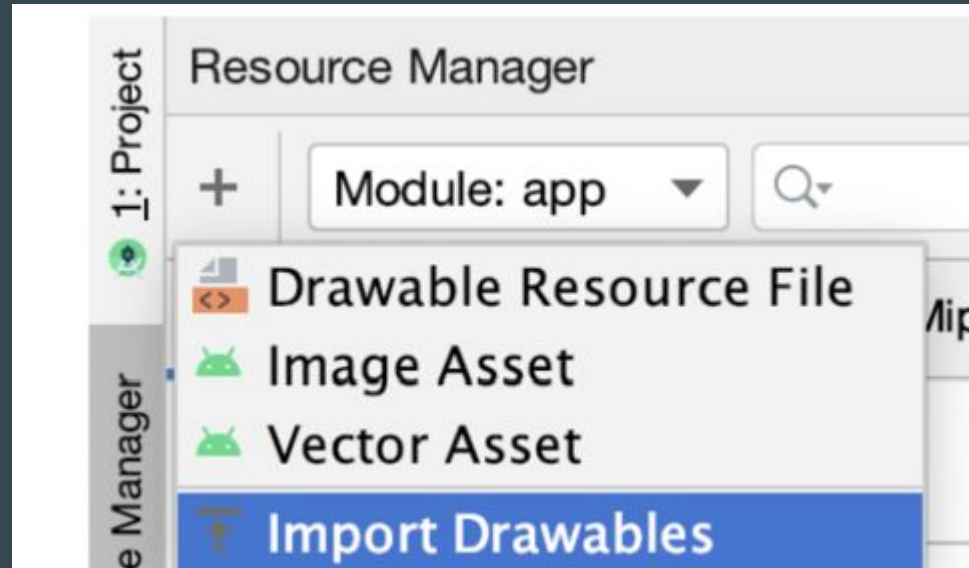


Image reference

once you are done importing images, you should be able to see those images in the Resource Manager

All these images can be referred in your Kotlin code with their resource Ids:

Example

```
R.drawable.shiba_calm
```


Navigation within an app

Navigation component

a collection of libraries, tooling, and IDE integrations for creating navigation paths through an app


works best with the paradigm of "one activity, many fragments" over having many activities

composed of three parts that work together

- Navigation graph
- Navigation Host (NavHost)
- Navigation Controller (NavController)

How to use the Navigation component?

add some dependencies to the build.gradle file.

```
dependencies {  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.5.3'  
     implementation 'androidx.navigation:navigation-ui-ktx:2.5.3'
```

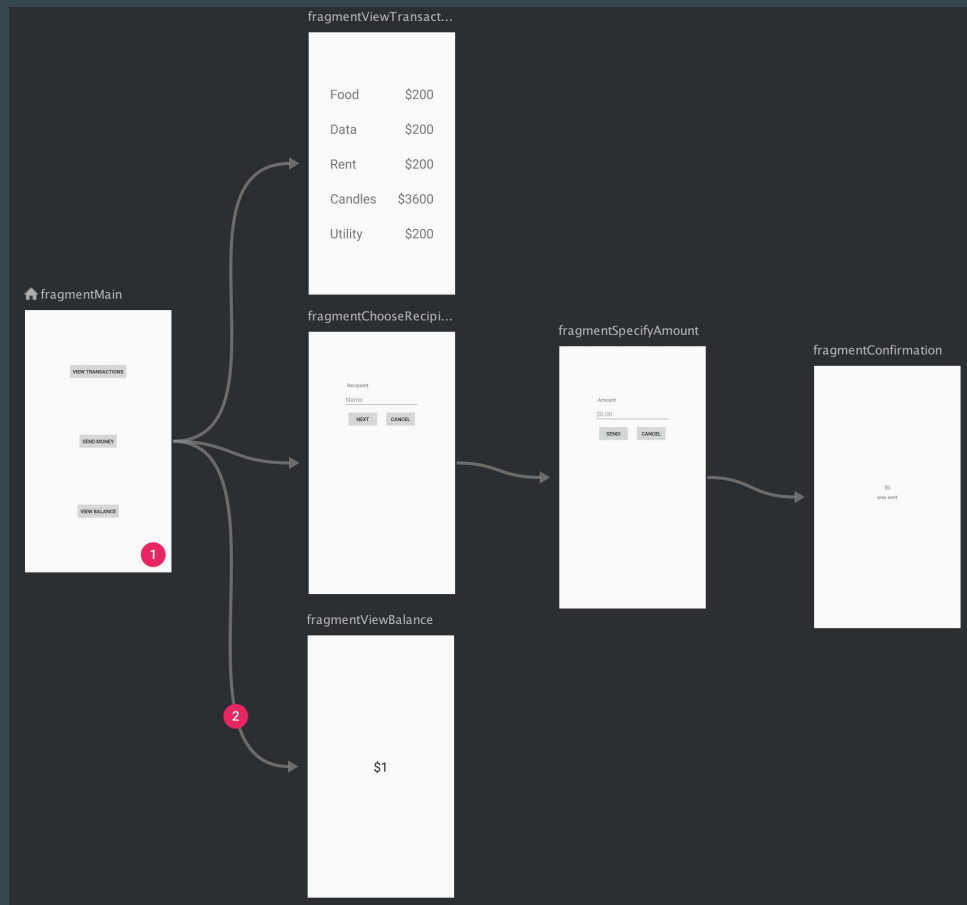
Navigation Graph

A navigation graph is a resource file that contains all of your destinations and actions

The graph represents all of your app's navigation paths.

TV Analogy

- a list of available programs to watch.



Navigation Host (NavHost)

an empty container where destinations are swapped in and out as a user navigates through your app

A navigation host must implement the NavHost interface

The default NavHost implementation, NavHostFragment, handles swapping fragment destinations for you

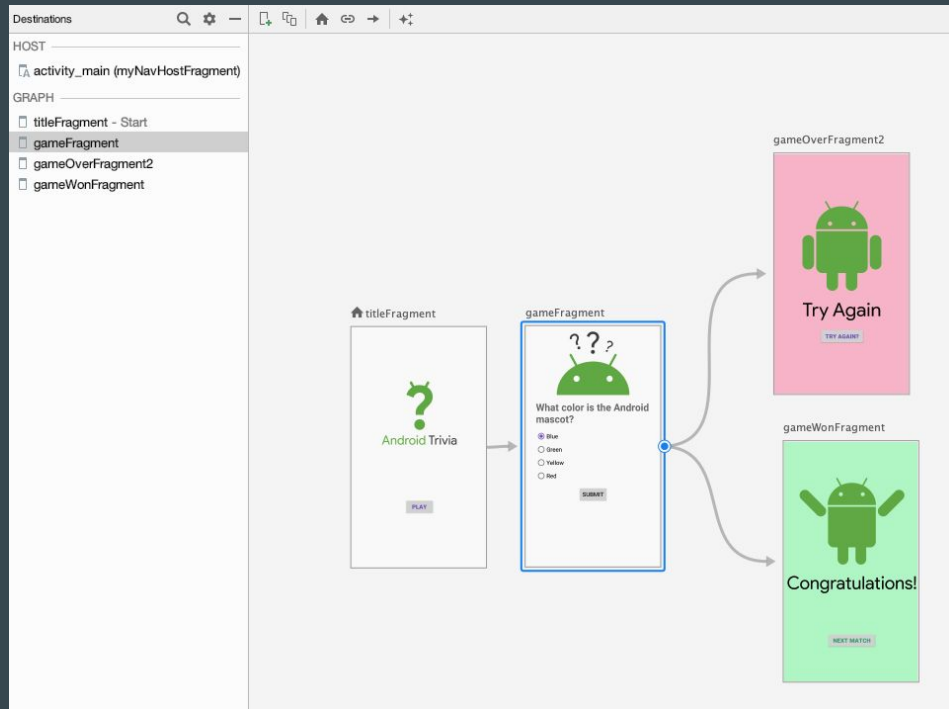
- android:name attribute has a value that is the fully qualified class name of our fragment
- app:defaultNavHost is set to true to ensure that this navigation host will intercept the system's Back button taps
- app:navGraph points to the navigation graph listing all of our fragment destinations

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

Navigation Editor

the Design view of the navigation graph resource file shows you a visual representation of the connections between different destinations for your Activity

can link destinations (the arrow button) and set entry points (the home button) from this interface



Creating a Fragment

To create a fragment, extend the `Fragment` class

In an Activity, we would override the `onCreate()` method

within a fragment's `onCreate()` method, it isn't guaranteed that any of the host activity's view hierarchy is initialized

layout inflation is done in the fragment's `onCreateView()` method

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

Specifying Fragment destinations

Fragment destinations are denoted by the action tag in the navigation graph.

Actions can be defined in XML directly, or using the Navigation Editor

Navigation Editor's navigation graph UI, specify destination paths by dragging from a source fragment to a destination fragment

automatically gives the action an ID in the form
action_<sourceFragment>_to_<destinationFragment>

```
<fragment
    android:id="@+id/welcomeFragment"
    android:name="com.example.android.navigation.WelcomeFragment"
    android:label="fragment_welcome"
    tools:layout="@layout/fragment_welcome" >

    <action
        android:id="@+id/action_welcomeFragment_to_detailFragment"
        app:destination="@id/detailFragment" />

</fragment>
```


Navigation Controller (NavController)

NavController manages UI navigation in a navigation host.

Specifying a destination path only names the action, but it doesn't execute it.

After naming all your paths, to actually navigate to them, you need to use the NavController when the button or link in your UI is selected.

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        val navController = findNavController(R.id.myNavHostFragment)  
    }  
  
    fun navigateToDetail() {  
        navController.navigate(R.id.action_welcomeFragment_to_detailFragment)  
    }  
}
```

Custom navigation behavior

Passing data between destinations

In many cases, your destination fragment will need some data to construct itself

The Navigation component has a Gradle plugin called Safe Args

Safe Args generates simple object and builder classes for type-safe navigation, and access to any associated arguments

recommended when navigating and passing data between destinations to ensure type-safety.

Generates a `<SourceDestination>Directions` class with methods for every action in that destination

Generates a class to set arguments for every named action

Generates a `<TargetDestination>Args` class providing access to the destination's arguments

Setting up Safe Args

edit two of your Gradle files

```
buildscript {  
    repositories {  
        google()  
    }  
    dependencies {  
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"  
    }  
}
```

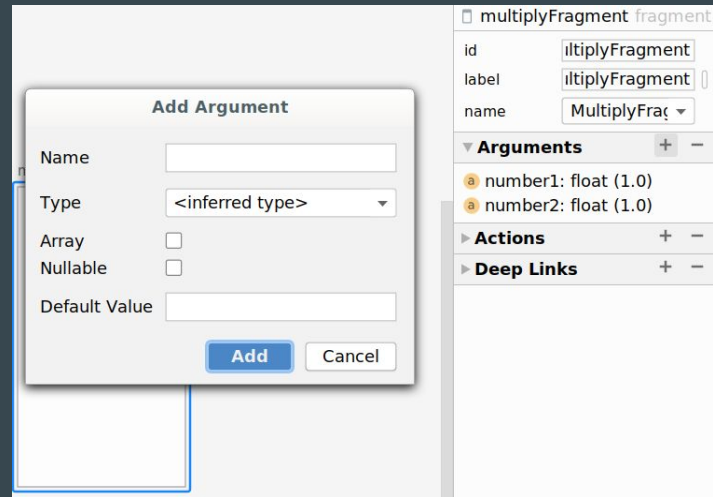
```
apply plugin: "androidx.navigation.safeargs.kotlin"
```

Sending data to a Fragment

After setting up the Gradle dependencies and plugins for Safe Args

the steps you must complete to send data between fragments

- Create arguments the destination fragment will expect.
- Create action to link from source to destination.
- Set the arguments in the action method on <Source>FragmentDirections.
- Navigate according to that action using the Navigation Controller.
- Retrieve the arguments in the destination fragment.



Destination arguments

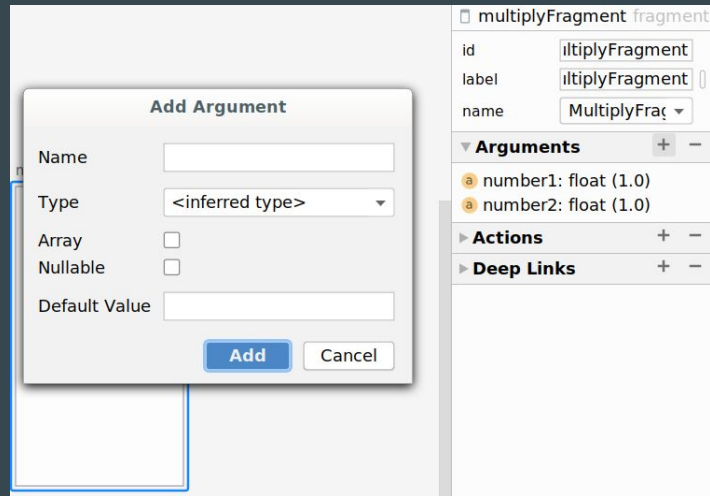
Define arguments on the destination fragment that will receive them

Android Studio provides a helpful way to create arguments for a fragment within the Navigation Editor

creates the XML code in our navigation graph for us

Example

- set two arguments (of type Float) that MultiplyFragment expects to receive



```
<fragment
    android:id="@+id/multiplyFragment"
    android:name="com.example.arithmetic.MultiplyFragment"
    android:label="MultiplyFragment" >
    <argument
        android:name="number1"
        app:argType="float"
        android:defaultValue="1.0" />
    <argument
        android:name="number2"
        app:argType="float"
        android:defaultValue="1.0" />
</fragment>
```



Supported argument types

can pass most types as arguments in a fragment

Primitive types like numbers, booleans, and strings require their lowercase name as the argument type

Enum classes require the fully qualified name

resources from the res/ directory require “reference” as the argument type

Type	Type Syntax app:argType=<type>	Supports Default Values	Supports Null Values
Integer	"integer"	Yes	No
Float	"float"	Yes	No
Long	"long"	Yes	No
Boolean	"boolean"	Yes ("true" or "false")	No
String	"string"	Yes	Yes
Array	above type + "[]" (for example, "string[]" "long[]")	Yes (only "@null")	Yes
Enum	Fully qualified name of the enum	Yes	No
Resource reference	"reference"	Yes	No

Supported argument types: Custom classes

Serialization is the process of taking an object's state and converting it into a stream of data for transmission

Serializable is the interface you would implement for a pure JVM class

Parcelable does serialization as well, but in a way that's more optimized for Android

will need to use one or the other to pass custom classes as arguments.

Type	Type Syntax <code>app:argType=<type></code>	Supports Default Values	Supports Null Values
Serializable	Fully qualified class name	Yes (only <code>"@null"</code>)	Yes
Parcelable	Fully qualified class name	Yes (only <code>"@null"</code>)	Yes

Create action from source to destination

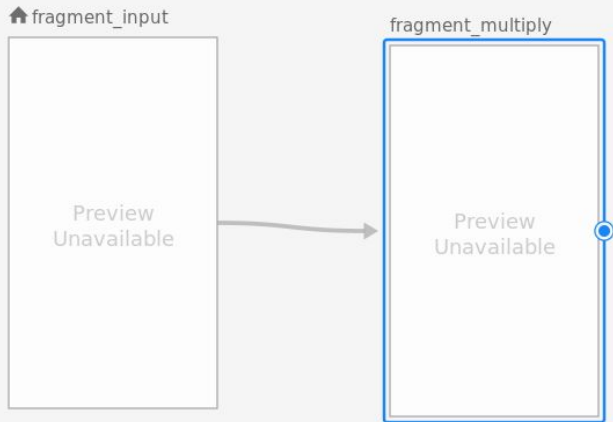
In our navigation graph (nav_graph.xml)

Can create an action from the InputFragment to the MultiplyFragment

```
<fragment
  android:id="@+id/fragment_input"
  android:name="com.example.arithmetic.InputFragment">

  <action
    android:id="@+id/action_to_multiplyFragment"
    app:destination="@id/multiplyFragment" />

</fragment>
```



Navigating with actions

call the action function on the Directions class
(called InputFragmentDirections)

the target destination (MultiplyFragment) requires
two arguments, the action class for that transition
also has two arguments

we can call navigate() on the NavController

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
    binding.button.setOnClickListener {  
        val n1 = binding.number1.text.toString().toFloatOrNull() ?: 0.0  
        val n2 = binding.number2.text.toString().toFloatOrNull() ?: 0.0  
  
        val action = InputFragmentDirections.actionToMultiplyFragment(n1, n2)  
        view.findNavController().navigate(action)  
    }  
}
```

Retrieving Fragment arguments

we are able to retrieve our arguments

MultiplyFragmentArgs is another generated class based on our navigation graph

navArgs comes from
androidx.navigation.fragment.navArgs

After the fragment's view is created, we can access and use the arguments

```
class MultiplyFragment : Fragment() {  
    val args: MultiplyFragmentArgs by navArgs()  
    lateinit var binding: FragmentMultiplyBinding  
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {  
        super.onCreateView(view, savedInstanceState)  
        val number1 = args.number1  
        val number2 = args.number2  
        val result = number1 * number2  
        binding.output.text = "${number1} * ${number2} = ${result}"  
    }  
}
```

Navigation UI

Menus revisited

learned how to navigate between Activities using intents

If the menu item id is the same as an action or destination id, NavigationUI will route to the proper place

frees us from having to handle each menu item separately

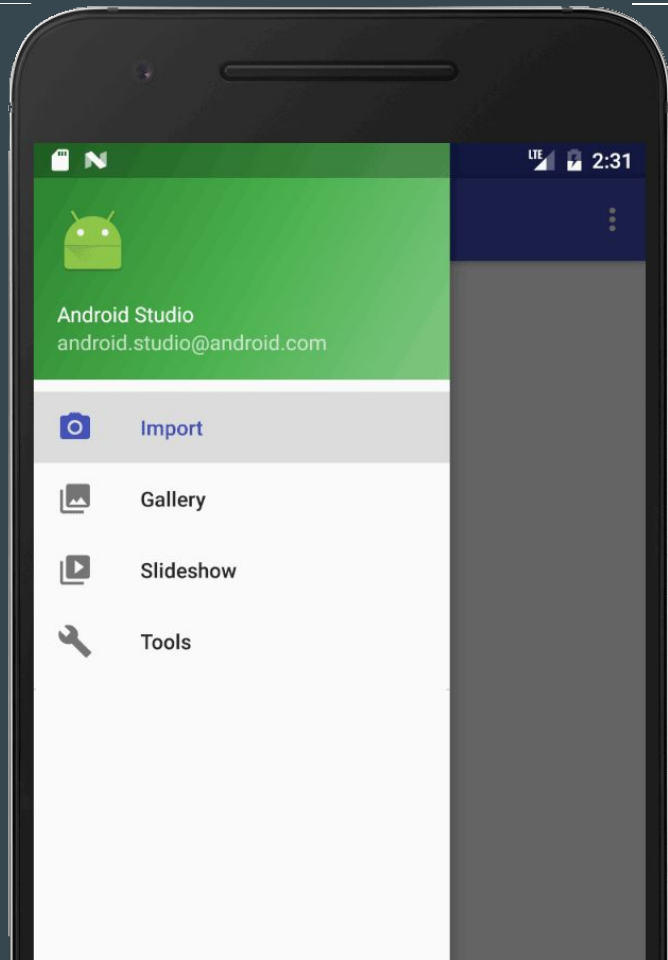
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    val navController = findNavController(R.id.nav_host_fragment)  
    return item.onNavDestinationSelected(navController) ||  
        super.onOptionsItemSelected(item)  
}
```

DrawerLayout for navigation drawer

Using a navigation drawer requires a bit of setup,

https://developer.android.com/guide/navigation/navigation-ui#add_a_navigation_drawer

declare a DrawerLayout as the root view. Within the DrawerLayout, add a layout for the main UI content (which is NavHostFragment in this case), and another view for the contents of the navigation drawer (which is the NavigationView).



Finish setting up navigation drawer

Within your Activity code, connect the DrawerLayout to the navigation graph

set up the NavigationView with the NavController

This will call MenuItem.onNavDestinationSelected when a menu item is selected

The selected item in the NavigationView will automatically be updated when the destination changes

```
val appBarConfiguration =  
AppBarConfig(navController.graph, drawer)
```

```
val navView =  
findViewById<NavigationView>(R.id.nav_view)  
  
navView.setupWithNavController(navController)
```

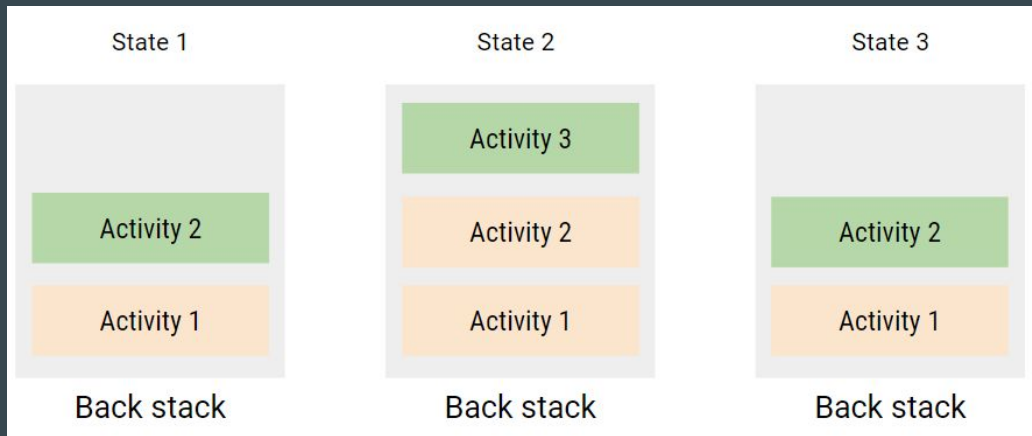
Understanding the back stack

concept that you'll encounter as you deal with app navigation

Android keeps track of the activities you've opened as a collection of activities in a stack

"last in, first out" approach for keeping track of the activities you've visited

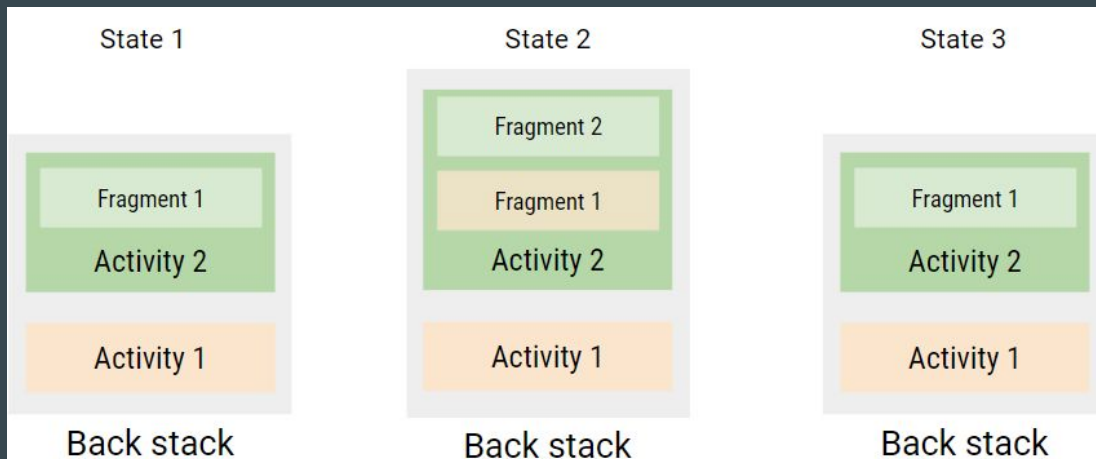
adds the most recent Activity you just started on top of the stack



Fragments and the back stack

when using fragments with the Navigation component, there is also a back stack of fragments

- **State 1:** We start with Activity 2 displaying Fragment 1. (Activity 1 is currently stopped and in the background.)
- **State 2:** Then within Activity 2, we navigate to another Fragment (Fragment 2), which gets added to the top of the back stack within its host Activity.
- **State 3:** When you tap the Back button, Fragment 2 is popped off the back stack and Fragment 1 would become the active Fragment again.



Activity Lifecycle

Why Activity Lifecycle Matters

important to know the states of the Activity lifecycle so that you can implement proper app behavior based on user expectations

Example

- Preserve user data and state if:
 - User temporarily leaves app and then returns
 - User is interrupted (for example, a phone call)
 - User rotates device

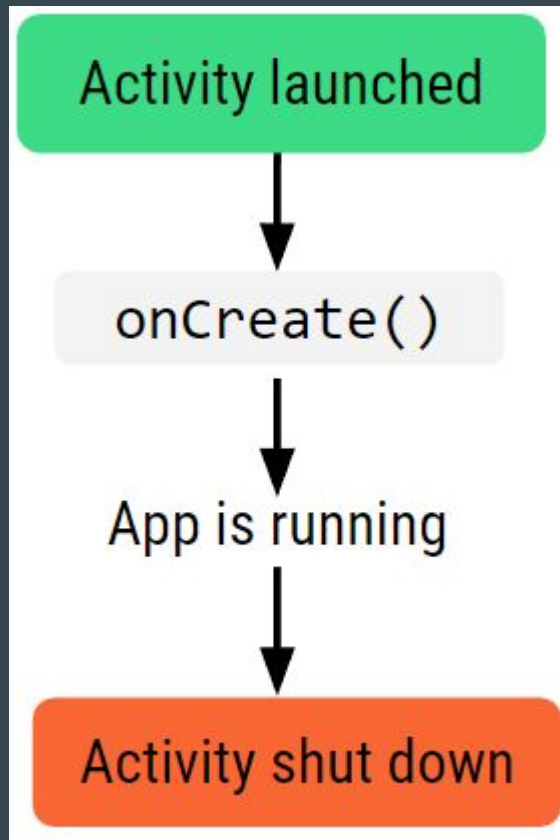
developer's responsibility to handle these state changes gracefully without crashing or wasting system resources

provides callback methods so you can know when the Activity is entering each state of the Activity lifecycle

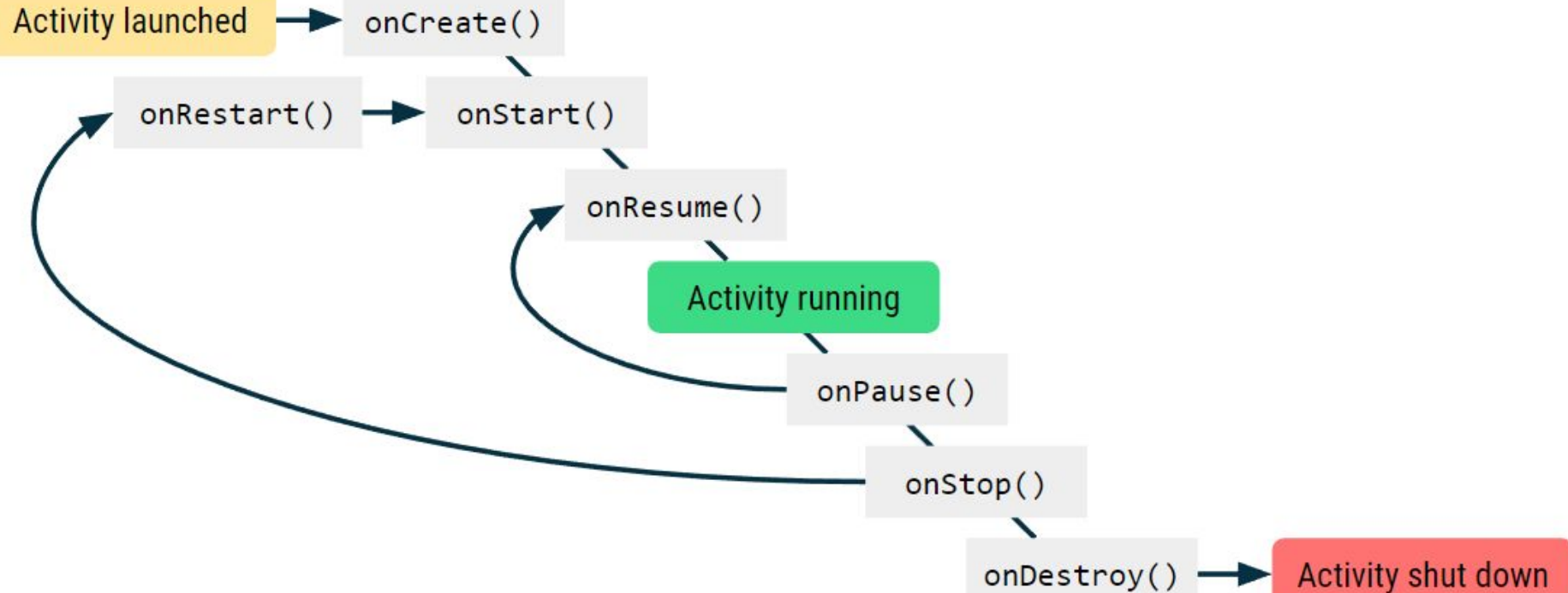
Simplified activity lifecycle

briefly talked about the Activity lifecycle

- Your Activity launches
- onCreate() is called
- some initialization happens in order for the activity to run successfully



Activity lifecycle in depth



Activity States

activity can transition between different state

mostly a linear process that moves forward with each state

As soon as the Activity is partially covered, or the user navigates away from the Activity, then we move through the Paused and Stopped states

Some states can make backward transitions

Activity class provides callbacks for these state changes.



onCreate()

we've overridden this callback method to do activity setup and layout inflation

onCreate() method is called when the system first creates your Activity

- Perform startup logic for your app
- inflating the activity's UI
- initializing any variables or components of your app.

Activity instance does not stay in the created state

After the onCreate() code is executed, the Activity moves into the started state and the system calls the onStart() method

onStart()

`onStart()` is called when the activity is started and becomes visible to the user called when the Activity is first started (coming from `onCreate()`), or restarted (`onRestart()`).

onResume()

When the Activity enters the resumed state, the onResume() callback is called

The Activity is now active and ready to receive input from the user

Stays in this state

- until the user (or system) does something that pauses the Activity,
- onPause() is called.

onPause()

The paused state is entered when the Activity has lost focus, but is still visible on the screen (The Activity may be in the process of being closed)

If another Activity is launched on top, then you will receive a call to onStop() after onPause()

if this Activity is only partially covered and then returns to the foreground, then the onResume() method could be called next.

onStop()

An activity that is stopped should release many of its resources because the activity is no longer visible to the user.

- stop refreshing UI
- running animations
- other visual changes

Activities in a stopped state are not gone

If the user navigates back to the Activity, then `onRestart()` will be called next

If the Activity is finishing or being destroyed by the system, then `onDestroy()` is called next.

onDestroy()

Activities can enter the destroyed state under these conditions

- the user has killed the app proactively
- the activity has finished
- configuration change
 - Rotated device
 - Single -> multi window mode
 - Multi -> single window mode

onDestroy() should handle final cleanup of resources.

Summary of activity states

State	Callbacks	Description
Created	<code>onCreate()</code>	Activity is being initialized.
Started	<code>onStart()</code>	Activity is visible to the user.
Resumed	<code>onResume()</code>	Activity has input focus.
Paused	<code>onPause()</code>	Activity does not have input focus.
Stopped	<code>onStop()</code>	Activity is no longer visible.
Destroyed	<code>onDestroy()</code>	Activity is destroyed.

Save state

In the `onDestroy()` callback slide, we talked about configuration changes. Configuration changes cause an Activity to be destroyed and recreated.

The framework lets us save a small amount of data in a Bundle to reconstruct the layout

- Activity is destroyed and restarted, or app is terminated and activity is started.
- Store user data needed to reconstruct app and activity Lifecycle changes:
 - Use Bundle provided by `onSaveInstanceState()`.
 - `onCreate()` receives the Bundle as an argument when activity is created again.

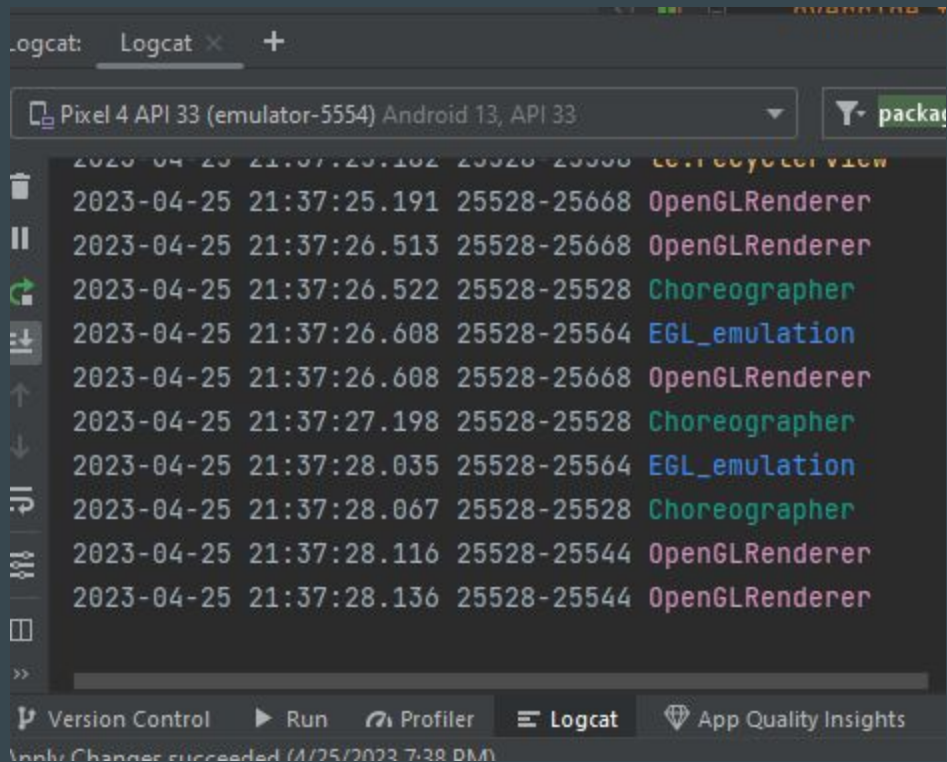
Logging

Logging in Android

Logging can be a useful way to better understand the Activity and Fragment lifecycle transitions

The Logcat window in Android Studio displays system messages, as well as Log messages that you've added to your app

If your app crashes, the stack trace can be seen in logcat, which is useful for debugging.



Write logs

can send messages to Logcat with different priority levels to indicate the importance of the message

Do not compile verbose logs into your app, except during development

Debug logs are compiled in, but stripped at runtime, while error, warning, and info logs are always kept.

Priority level	Log method
Verbose	<code>Log.v(String, String)</code>
Debug	<code>Log.d(String, String)</code>
Info	<code>Log.i(String, String)</code>
Warning	<code>Log.w(String, String)</code>
Error	<code>Log.e(String, String)</code>

Framework lifecycle

Fragment lifecycle

a fragment represents a behavior or portion of the UI and can be thought of as a "sub-activity."

A fragment must always be hosted in an Activity

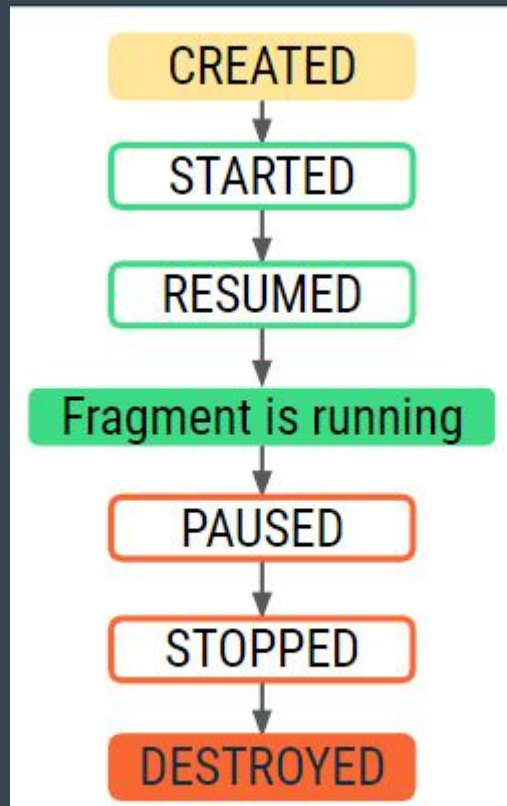
the fragment's lifecycle is directly affected by the host Activity's lifecycle.

- If an Activity loses focus and is stopped or destroyed, any hosted fragments will be stopped or destroyed as well
- While an Activity is resumed, you can add or remove fragments from it.

Fragment states

a lot of callback methods that are similar to the Activity class

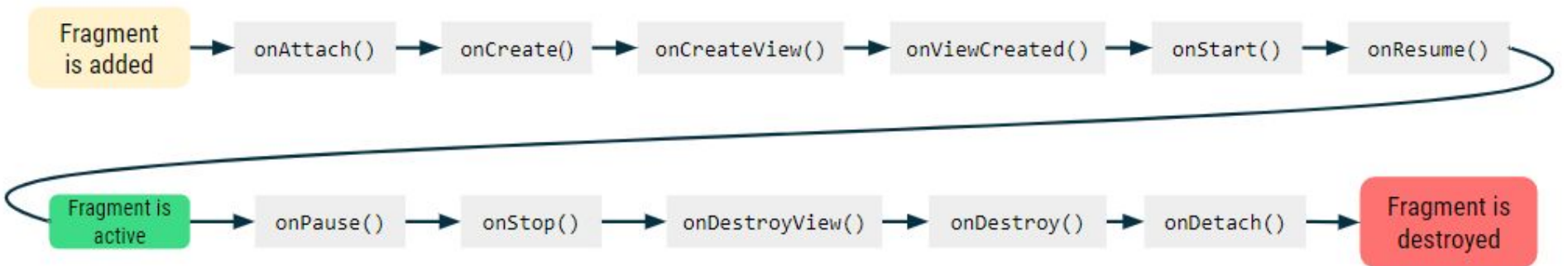
There are some new callbacks (on the next slide)



Fragment lifecycle diagram

New fragment callbacks

- `onAttach()`
- `onCreateView()`
- `onViewCreated()`
- `onDestroyView()`
- `onDetach()`



onAttach()

onAttach() immediately precedes a fragment's onCreate() method

onAttach() is called before the fragment has access to its layout

won't be overriding this method very often.

onCreateView()

called to have the fragment instantiate its UI view.

Called to create the view hierarchy associated with the fragment

called between the onCreate() and onViewCreated() methods

Inflate the fragment layout here and return the root view

onViewCreated()

Called when view hierarchy has already been created

Perform any remaining initialization here

- Ex. restore state from Bundle

Can finish setting up the UI because we can be sure that views are available at this point.

`onDestroyView()` and `onDetach()`

`onDestroyView()` is called when view hierarchy of fragment is removed.

`onDetach()` is called when fragment is no longer attached to the host.

Summary of fragment states

State	Callbacks	Description
Initialized	<code>onAttach()</code>	Fragment is attached to host.
Created	<code>onCreate()</code> , <code>onCreateView()</code> , <code>onViewCreated()</code>	Fragment is created and layout is being initialized.
Started	<code>onStart()</code>	Fragment is started and visible.
Resumed	<code>onResume()</code>	Fragment has input focus.
Paused	<code>onPause()</code>	Fragment no longer has input focus.
Stopped	<code>onStop()</code>	Fragment is not visible.
Destroyed	<code>onDestroyView()</code> , <code>onDestroy()</code> , <code>onDetach()</code>	Fragment is removed from host.

Save fragment state across config changes

Preserve UI state in fragments by storing state in Bundle:

- `onSaveInstanceState(outState: Bundle)`

Retrieve that data by receiving the Bundle in these fragment callbacks:

- `onCreate()`
- `onCreateView()`
- `onViewCreated()`

Lifecycle-aware components

Lifecycle-aware components

Adjust their behavior based on activity or fragment lifecycle

- Use the `androidx.lifecycle` library
- Lifecycle tracks the lifecycle state of an activity or fragment
 - Holds current lifecycle state
 - Dispatches lifecycle events (when there are state changes)

LifecycleOwner

Interface that says this class has a lifecycle

Implementers must implement `getLifecycle()` method

Example

This interface abstracts the ownership of a Lifecycle from individual classes,

- Fragment
- AppCompatActivity

LifecycleObserver

For any class that wants to listen for lifecycle events, implement the LifecycleObserver interface

uses the annotated OnLifecycleEvent methods

register the observer with the Lifecycle to get notified of lifecycle events

lifecycle-aware components will help you write cleaner and better organized code that's easier to maintain

```
class MyObserver : LifecycleObserver {  
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
    fun connectListener() {  
        ...  
    }  
}
```

```
myLifecycleOwner.getLifecycle().addObserver(MyObserver())
```

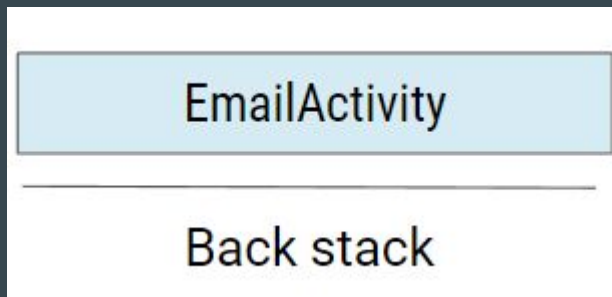
Tasks and back stack

Back stack of activities

open up an email app on your phone

- if no task exists for the app, a new task is created

The main activity of this email app is added to the back stack. It's on top of the back stack



Add to the back stack

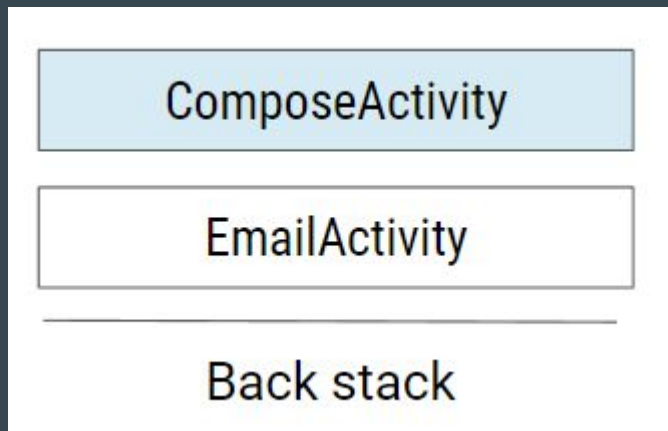
write a new email

opening up a second activity (ComposeActivity)

The first activity (EmailActivity) has been paused and stopped, in terms of lifecycle states

ComposeActivity has been created, started, and resumed since you're interacting with it

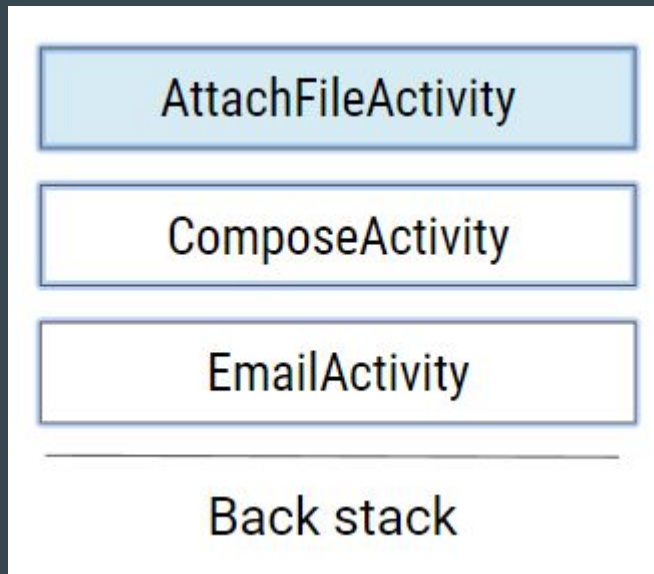
the ComposeActivity is now on top of the back stack



Add to the back stack again

decide to attach a photo to this email

- click on the "Attach File" menu option, which opens up a new activity to browse the files on your device
- The ComposeActivity is no longer visible, so that activity is stopped
- The AttachFileActivity is now active and is on the top of the back stack.

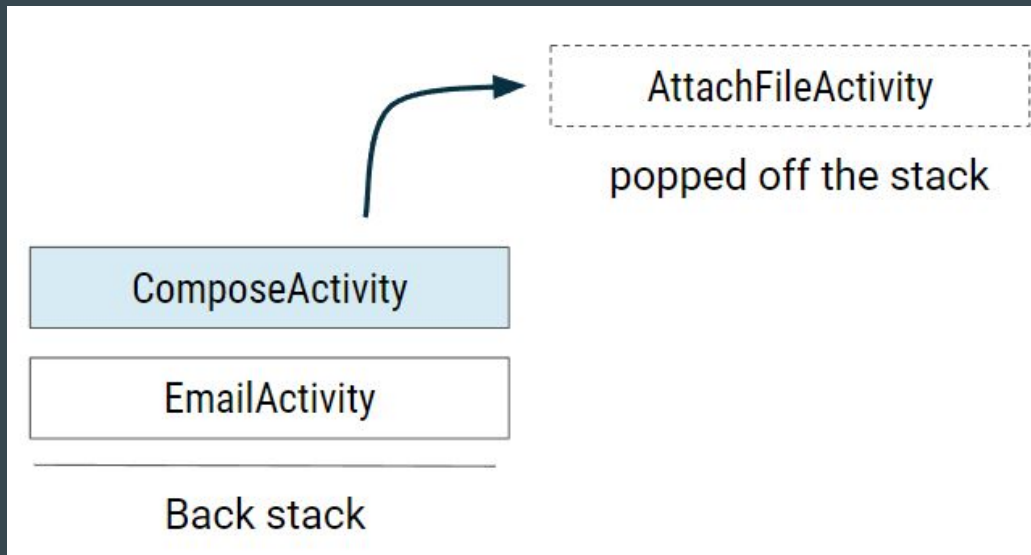


Tap Back button

decides to tap the Back button

the AttachFileActivity is popped off the stack

the ComposeActivity is started again and resumed, and becomes the activity that the user is interacting with



Tap Back button again

ComposeActivity is popped off the back stack

EmailActivity is left on top and becomes visible and active again

