Fregment

In Android, a `Fragment` is a modular and reusable component that represents a portion of a user interface or behavior within an activity. Fragments were introduced to support the development of flexible and responsive user interfaces, especially for larger screens like tablets. Fragments have their own lifecycle and can be combined to create multi-pane UIs.

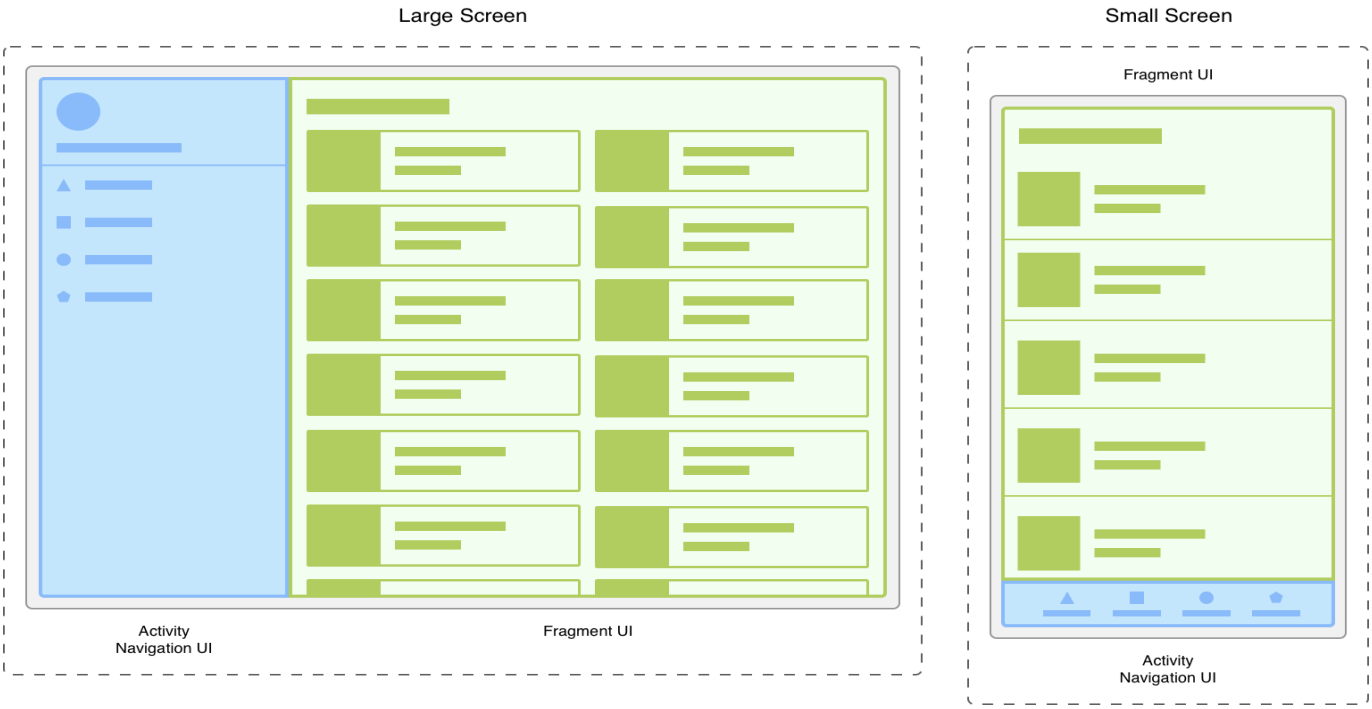


Figure 1. Two versions of the same screen on different screen sizes. On the left, a large screen contains a navigation drawer that is controlled by the activity and a grid list that is controlled by the fragment. On the right, a small screen contains a bottom navigation bar that is controlled by the activity and a linear list that is controlled by the fragment.

Dividing your UI into fragments makes it easier to modify your activity's appearance at runtime. While your activity is in the STARTED lifecycle state or higher, fragments can be added, replaced, or removed. And you can keep a record of these changes in a back stack that is managed by the activity, so that the changes can be reversed.

Fragments in Android are used to represent a behavior or a portion of user interface within an activity. They are like modular sections of an activity that can be combined or reused in different activities. Fragments are particularly useful in various scenarios, including:

1. UI Modularization:

Break down a complex UI into smaller, manageable components. Each fragment can represent a part of the UI, and these fragments can be combined in different ways within various activities.

2. Tablet and Large Screen Layouts:

In tablet or landscape mode, you might want to display multiple fragments side by side. Fragments help in creating a responsive UI that adapts to different screen sizes.

3. Reusability:

Fragments can be reused in multiple activities. This promotes code reusability and helps maintain a consistent user interface across different parts of your application.

4. Dynamic UI:

Fragments are useful for creating dynamic and flexible user interfaces. You can dynamically add, remove, or replace fragments based on user interactions or other runtime conditions.

5. Multi-Pane Layouts:

In multi-pane layouts, such as a master-detail view, fragments can be used to represent both the master and detail portions. This is common in tablet interfaces.

6. Navigation Drawer:

Fragments are often used with navigation drawers to switch between different sections or functionalities of an app.

7. Tabbed Interfaces:

Each tab in a tabbed interface can be implemented as a fragment, allowing users to switch between different content views.

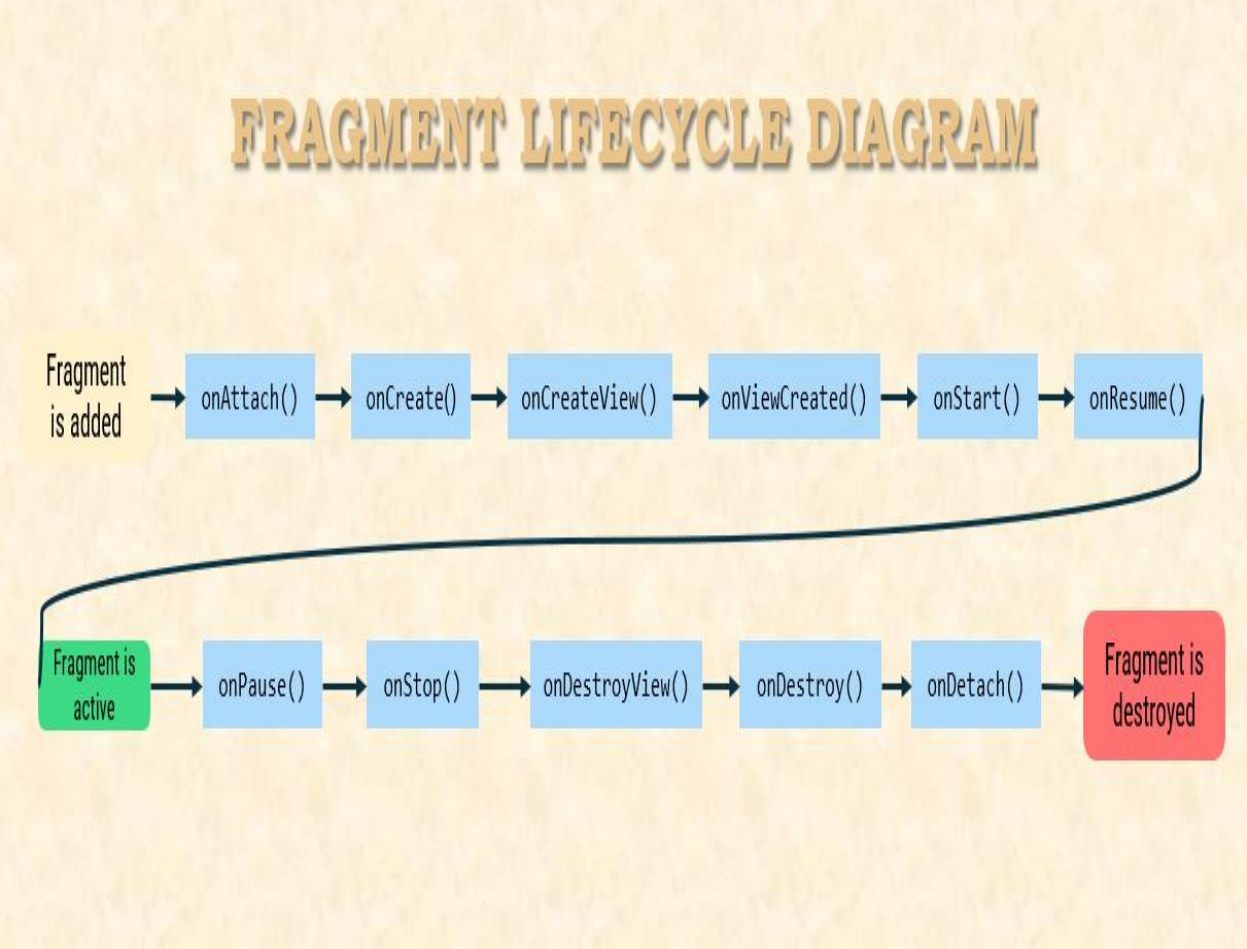
8. Communication Between UI Components:

Fragments can communicate with each other through their parent activity. This is useful for passing data or events between different parts of the UI.

9. Code Organization:

Fragments provide a way to organize code related to a specific part of the user interface. This makes the codebase more modular and easier to maintain.

Fragment Lifecycle:



Fragments have their own lifecycle methods, similar to activities. Some of the key lifecycle methods include:

1) **onAttach():**
   Called when the fragment has been associated with the activity.
2) **onCreate():**
   Called to do initial creation of the fragment.
3) **onCreateView():**
   Called to create the UI for the fragment.
4) **onActivityCreated():**
   Called when the activity's `onCreate` method has returned.
5) **onStart():**
   Called when the fragment becomes visible to the user.

6) **onResume():**
   Called when the fragment is visible and actively running.
7) **onPause():**
   Called when the fragment is no longer interacting with the user.
8) **onStop():**
   Called when the fragment is no longer visible to the user.
9) **onDestroyView():**
   Called when the view hierarchy associated with the fragment is being removed.
10) **onDestroy():**
   Called when the fragment is no longer in use.
11) **onDetach():**
   Called when the fragment is no longer associated with the activity.

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

Fragment lifecycle Method role and purpose

1. onAttach(Context context)
Role: Called when the fragment is attached to its hosting activity.
Purpose: Allows the fragment to get a reference to the hosting activity and perform any necessary initialization.

2. onCreate(Bundle savedInstanceState)
Role: Called when the fragment is created.
Purpose: Initialization tasks, such as setting up the fragment's user interface and initializing variables. The `savedInstanceState` parameter is used to restore the fragment's previous state if applicable.

3. onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
Role: Called to create the fragment's view hierarchy.
Purpose: Inflates the fragment's layout and returns the root view. This is where the fragment's UI components are created.

4. onActivityCreated(Bundle savedInstanceState)
Role: Called when the hosting activity's `onCreate()` method has completed.
Purpose: Allows the fragment to access the activity's views and perform one-time initialization tasks that require the activity to be fully created.

5. onStart()
Role: Called when the fragment becomes visible to the user.
Purpose: Performs any tasks that should occur when the fragment starts, such as preparing to interact with the user.

6. onResume()
Role: Called when the fragment is about to start interacting with the user.
Purpose: Resumes tasks that were paused or stopped, such as updating the UI and acquiring resources.

7. onPause()

Role: Called when the fragment is no longer in the foreground and the user is not interacting with it.

Purpose: Pauses ongoing tasks and releases resources to ensure a smooth transition between fragments.

8. onStop()

Role: Called when the fragment is no longer visible to the user.

Purpose: Stops or releases resources that are not needed when the fragment is not in the foreground.

9. onDestroyView()

Role: Called when the fragment's view hierarchy is being destroyed.

Purpose: Cleans up resources associated with the fragment's UI components.

10. onDestroy()

Role: Called when the fragment is being destroyed.

Purpose: Performs cleanup tasks and releases resources before the fragment is removed from memory.

11. onDetach()

Role: Called when the fragment is detached from its hosting activity.

Purpose: Allows the fragment to release references to the activity and perform final cleanup.

Why use ViewModel class with fragment

Using a `ViewModel` class with a `Fragment` in Android is a recommended practice to separate concerns and improve the architecture of your app. Here are some reasons why you might want to use a `ViewModel` class with a `Fragment`:

1. Lifespan Management:

- **Fragments have a complex lifecycle, and they can be destroyed and recreated during configuration changes (like screen rotation). A `ViewModel` survives these changes because it is associated with the `ViewModelStore`, not the `Fragment` itself. This allows data to be preserved across configuration changes without the need to explicitly save and restore it.**

2. Separation of Concerns:

- **The `ViewModel` class helps in separating the UI-related data and logic from the `Fragment`. This is beneficial for maintaining a clean and modular codebase, making it easier to understand and maintain.**

3. Sharing Data Between Fragments:

- **`ViewModels` can be shared between multiple fragments. This is useful when you have data that needs to be shared or accessed by different parts of your app.**

4. Persistence and Caching:

- **You can use `ViewModel` to cache and manage data that should survive beyond the lifecycle of a single `Fragment`. This can be useful for handling data that should persist across different screens or user interactions.**

5. Testing:

- **`ViewModel` classes can be easily tested in isolation from the UI components, making it easier to write unit tests for your business logic.**

This allows the UI to be updated whenever the underlying data changes, and the data is retained across configuration changes.

For example :

Imagine you have a screen (a `Fragment` in Android) that shows some data, and this data can change or be updated based on user interactions. Now, the screen might get destroyed and recreated due to various reasons like rotating the device.

Here's the problem: If you just store that data directly in the `Fragment`, it might get lost when the `Fragment` is recreated. And then, you'd have to do some complicated stuff to save and restore that data every time the screen changes.

Now, enter the `ViewModel`. It's like a smart container that holds your data, but it doesn't get destroyed when the screen is recreated. So, you can put your data in the `ViewModel`, and it stays there even when the `Fragment` gets recreated.

In simple terms:

1. Data Survives Rotation:

- **`ViewModel` helps you keep your data when the phone is rotated or when there are other configuration changes. Without a `ViewModel`, your data might get lost during these changes.**

2. Separation of Concerns:

- **`ViewModel` helps you keep your code organized. It's like a place to put all the important stuff related to your data and business logic. Your `Fragment` can focus on how to display things, and the `ViewModel` handles the data part.**

3. Easy Communication:

- **If you have multiple `Fragments` or even different parts of your app, `ViewModel` helps them talk to each other. One `Fragment` can update the `ViewModel`, and others can listen for changes.**

In a nutshell, the `ViewModel` helps you keep your data safe and organized, making it easier to deal with the complex life of Android components.