**Unit-4**

### ****Working with View in Android:-****

In Android, a **View** is an object that represents a user interface element or UI component. Views are responsible for displaying content and receiving user interactions (e.g., touches, clicks, text input). Everything you see and interact with on an Android screen is a type of View.

### ****Types of Views in Android:****

1. **Text View**:
   * Displays text on the screen.
   * Example: A label or heading.
2. **Button**:
   * A clickable UI element.
   * Example: A button that triggers an action when clicked.
3. **Image View**:
   * Displays an image or graphic.
   * Example: A logo or an icon.
4. **Edit Text**:
   * Allows users to enter text input.
   * Example: A field where users can type their name or email.
5. **Check Box**:
   * A UI element that allows users to select or deselect an option.
   * Example: "I agree to the terms and conditions".
6. **Radio Button**:
   * Similar to a checkbox, but used within a group where only one option can be selected at a time.
   * Example: Select a gender (Male/Female/Other).
7. **Image Button**:
   * A button that displays an image instead of text.
   * Example: A "play" button with an icon.

### ****View Groups:****

Views can be arranged in containers called **View Groups**. These are special types of Views that hold other Views and control their layout on the screen. Some common ViewGroups are:

* **Linear Layout**: Aligns its children (views) either vertically or horizontally.
* **Relative Layout**: Positions its children relative to one another.
* **Frame Layout**: Displays a single child view in a stack (ideal for showing one child at a time).

### ****View Hierarchy:****

* Views are organized in a hierarchy where each View can contain other Views or ViewGroups.
* The **root view** is the highest-level container, and all other Views are nested under it.
* For example, a LinearLayout might be the root view, and it may contain a Button, TextView, and ImageView.

### ****Inflating Views:****

* Views can be defined in XML files (layout files) using tags such as <TextView>, <Button>, etc.
* These XML files are then **inflated** into actual View objects by the Android framework.

### ****How Views Work:****

1. **Display Content**: Views are responsible for rendering content on the screen. For instance, TextView renders text, and ImageView renders images.
2. **Handle User Input**: Views can respond to user actions, such as clicks, touches, or typing.
   * For example, a Button will execute a piece of code when clicked.
3. **Layout and Positioning**: Views are arranged within a layout (ViewGroup). The layout defines how the Views will be positioned on the screen.
4. **Interaction and Events**: Views listen for user interactions like clicks, gestures, and text changes. These interactions trigger events (such as onClick() or onTouch()), which can be handled by defining event listeners in your code.

### ****Attributes of Views:****

* Each View has certain attributes that control its appearance and behavior.
  + android:layout\_width: Specifies the width of the View.
  + android:layout\_height: Specifies the height of the View.
  + android:text: Sets the text for TextView or Button.
  + android:onClick: Specifies the method to be called when a Button is clicked.

### ****Example of a Simple Layout (XML):****

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, World!" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Click Me" />

</LinearLayout>

In the above example, a LinearLayout holds a TextView and a Button.

### ****Conclusion:****

* **Views** are the essential building blocks for creating UI in Android.
* **ViewGroups** are containers that hold and manage the arrangement of other Views.
* Views can be interacted with by the user, and they display content on the screen, handle touch input, and trigger events.
* Views are usually defined in XML files but can also be created programmatically.

### ****Fragment in Android?****

A **Fragment** is a reusable, self-contained piece of a user interface (UI) that can be combined with other fragments in an activity to create flexible and dynamic UIs. Fragments are like mini-activities, with their own lifecycle, views, and logic, but they must always be hosted within an Activity.

### ****Why Use Fragments?****

1. **Modularity**: Fragments allow you to break down a complex UI into smaller, manageable parts. You can reuse fragments in different activities, improving code reuse.
2. **Responsive UIs**: They make it easier to create flexible UIs that adapt to different screen sizes and orientations, such as tablets and phones.
3. **Dynamic UI**: You can add, remove, or replace fragments dynamically at runtime, which is useful in scenarios like navigation or changing content without restarting the activity.

### ****Fragment Lifecycle:****

Fragments have their own lifecycle that is closely tied to the activity’s lifecycle. Here's a simplified lifecycle for a fragment:

1. **onAttach()**: Called when the fragment is attached to its parent activity.
2. **onCreate()**: Called when the fragment is being created. You can initialize resources here.
3. **onCreateView()**: Called to inflate the fragment's layout (UI). You define the fragment's UI here.
4. **onActivityCreated()**: Called after the activity’s onCreate() has been completed. Useful for final initialization.
5. **onStart()**: Called when the fragment is becoming visible to the user.
6. **onResume()**: Called when the fragment becomes active and starts interacting with the user.
7. **onPause()**: Called when the fragment is no longer interacting with the user.
8. **onStop()**: Called when the fragment is no longer visible.
9. **onDestroyView()**: Called when the fragment’s view is destroyed. Cleanup can be done here.
10. **onDestroy()**: Called when the fragment is being destroyed.
11. **onDetach()**: Called when the fragment is detached from the activity.

### ****Fragment Transaction:****

Fragments cannot exist alone—they need to be added, replaced, or removed in an activity through **Fragment Transactions**. A **FragmentTransaction** is used to manage the addition, removal, and replacement of fragments in an activity dynamically.

Basic operations with FragmentTransactions:

* **add()**: Adds a fragment to the activity.
* **replace()**: Replaces an existing fragment with a new one.
* **remove()**: Removes an existing fragment from the activity.
* **hide()** and **show()**: Hide or show fragments without removing them.

Example of adding a fragment dynamically:

FragmentTransaction transaction = getFragmentManager().beginTransaction();

transaction.add(R.id.fragment\_container, new MyFragment());

transaction.commit();

### ****Types of Fragments:****

1. **Static Fragments**: Fragments declared statically in XML layout files. They are added at the time of the activity's layout creation.
   * Example: Declaring a fragment in the XML layout file of an activity.
2. <fragment
3. android:id="@+id/myFragment"
4. android:name="com.example.MyFragment"
5. android:layout\_width="match\_parent"
6. android:layout\_height="match\_parent" />
7. **Dynamic Fragments**: Fragments added at runtime using FragmentTransaction, allowing you to modify the activity’s UI dynamically.
   * Example: Adding a fragment dynamically in Java code.
8. FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
9. transaction.replace(R.id.fragment\_container, new MyFragment());
10. transaction.commit();

### ****Communicating Between Fragments and Activities:****

* **Activity to Fragment**: You can pass data from an activity to a fragment using the fragment's arguments, typically via a Bundle.
* Bundle bundle = new Bundle();
* bundle.putString("key", "value");
* MyFragment fragment = new MyFragment();
* fragment.setArguments(bundle);
* **Fragment to Activity**: To communicate back from a fragment to its host activity, you can define a callback interface and have the activity implement it.
* public interface OnFragmentInteractionListener {
* void onFragmentInteraction(String data);
* }

### ****Fragment Layout and Views:****

In **onCreateView()**, you define the layout of the fragment by inflating a layout XML file (or creating it programmatically).

Example of inflating a layout for a fragment:

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_my, container, false);

}

### ****Advantages of Using Fragments:****

* **Flexibility**: Fragments make it easier to design UIs that work well on both small and large screens, such as phones and tablets.
* **Reusability**: Fragments can be reused across different activities, making the app more modular.
* **Dynamic UI**: You can replace, add, or remove fragments at runtime, which makes the app UI adaptable.

### ****When to Use Fragments:****

* When you need to create flexible layouts that adjust for different screen sizes (e.g., phones vs. tablets).
* When you want to break down complex UIs into smaller, reusable parts.
* When you want to implement UI elements that change dynamically at runtime.

### ****Conclusion:****

Fragments are a powerful component in Android development that helps create modular, reusable, and flexible UIs. They allow developers to build responsive layouts that adjust to various screen sizes and orientations, and enable dynamic content updates without reloading the entire activity. Understanding the fragment lifecycle and how to manage fragments within an activity is key to leveraging their full potential.

### GridView, WebView, ScrollView, ListView, RecyclerView:-

### ****1. GridView:****

**Theory:**

* A GridView is a view in Android that displays items in a two-dimensional grid (rows and columns).
* It's commonly used for displaying a collection of items such as images or texts in a grid format.

**Syntax:**

<GridView

android:id="@+id/gridView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:numColumns="auto\_fit"

android:columnWidth="80dp"

android:verticalSpacing="10dp"

android:horizontalSpacing="10dp" />

**Basic Usage in Java (Activity):**

GridView gridView = findViewById(R.id.gridView);

String[] items = {"Item 1", "Item 2", "Item 3", "Item 4"};

ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple\_list\_item\_1, items);

gridView.setAdapter(adapter);

### ****2. WebView:****

**Theory:**

* A WebView allows you to display web pages inside your Android application.
* You can load HTML content, URLs, or even execute JavaScript.

**Syntax:**

<WebView

android:id="@+id/webView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

**Basic Usage in Java (Activity):**

WebView webView = findViewById(R.id.webView);

webView.loadUrl("https://www.example.com");

### ****3. ScrollView:****

**Theory:**

* A ScrollView is used to enable vertical scrolling of a single child element.
* It only supports vertical scrolling; for horizontal scrolling, use HorizontalScrollView.

**Syntax:**

<ScrollView

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<LinearLayout

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<!-- Your content here -->

</LinearLayout>

</ScrollView>

**Basic Usage in Java (Activity):**

ScrollView scrollView = findViewById(R.id.scrollView);

### ****4. ListView:****

**Theory:**

* A ListView is a view that displays a vertically scrollable list of items.
* It is commonly used for lists where each item can be customized with layouts like images or text.

**Syntax:**

<ListView

android:id="@+id/listView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

**Basic Usage in Java (Activity):**

ListView listView = findViewById(R.id.listView);

String[] items = {"Item 1", "Item 2", "Item 3"};

ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple\_list\_item\_1, items);

listView.setAdapter(adapter);

### ****5. RecyclerView:****

**Theory:**

* A RecyclerView is a more advanced and flexible version of ListView and GridView. It supports both vertical and horizontal scrolling and more.
* It is designed to display large data sets efficiently by reusing views.

**Syntax:**

<androidx.recyclerview.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

**Basic Usage in Java (Activity):**

RecyclerView recyclerView = findViewById(R.id.recyclerView);

RecyclerView.Adapter adapter = new MyAdapter(data);

recyclerView.setAdapter(adapter);

### ****Final Example combining all views (GridView, WebView, ScrollView, ListView, RecyclerView):****

<ScrollView

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<LinearLayout

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<GridView

android:id="@+id/gridView"

android:layout\_width="match\_parent"

android:layout\_height="200dp"

android:numColumns="auto\_fit"

android:columnWidth="80dp" />

<WebView

android:id="@+id/webView"

android:layout\_width="match\_parent"

android:layout\_height="200dp" />

<ListView

android:id="@+id/listView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

<androidx.recyclerview.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

</LinearLayout>

</ScrollView>

**Comparison of the mentioned topics in a tabular form:**

| **Feature** | **GridView** | **WebView** | **ScrollView** | **ListView** | **RecyclerView** |
| --- | --- | --- | --- | --- | --- |
| **Purpose** | Displays items in a 2D grid (rows & columns). | Displays web pages or HTML content in the app. | Allows for vertical scrolling of content. | Displays a list of items with scrolling. | Displays large datasets efficiently with flexible layouts. |
| **Layout Type** | Grid layout (rows & columns). | Web content rendered inside the app. | Single child that can scroll (usually vertical). | List of items (one item per row). | List or grid of items with flexible item layouts. |
| **Use Cases** | Image galleries, icon grids, etc. | Rendering HTML, JavaScript, web-based content. | Content that might not fit within the screen. | Displaying long lists of static or dynamic items. | Large datasets with dynamic data, improving performance. |
| **Performance** | May have performance issues with large data. | Depends on web page; can consume a lot of resources. | No performance impact; just adds scrolling. | Slower with large datasets (needs more optimization). | Better performance with large datasets due to recycling views. |
| **Customizability** | Limited to grid structure and adapter. | Limited to web content, but can execute JavaScript. | Limited to scroll behavior and layout of child. | Customizable using adapters and custom views. | Highly customizable with various layout managers (Grid, Linear). |
| **Scrollability** | Horizontal/vertical scrolling, depending on the setup. | Scrolls content horizontally or vertically (if needed). | Vertical scrolling only. | Vertical scrolling. | Both horizontal and vertical scrolling. |
| **Adapter Needed** | Yes, to bind data to the grid. | No adapter; data is web-based content. | No adapter needed; just view hierarchy inside. | Yes, uses an adapter for binding data. | Yes, requires an adapter for binding data. |
| **Flexibility** | Less flexible (grid-based only). | Limited to web content. | Flexible in terms of child views inside. | Fairly flexible with custom views per item. | Highly flexible with different view types and layouts. |
| **View Recycling** | No view recycling. | No recycling. | No view recycling. | No recycling (basic list item view). | Yes, uses view holders and recycling for efficiency. |
| **Example Code** | <GridView> | <WebView> | <ScrollView> | <ListView> | <RecyclerView> |

### Key Differences:

* **GridView** is grid-based and does not support view recycling.
* **WebView** displays web content and doesn't need an adapter.
* **ScrollView** only provides vertical scrolling for its child views.
* **ListView** is a simple, vertical list with an adapter but lacks advanced optimizations like recycling.
* **RecyclerView** is the most advanced and flexible, with better performance and support for complex data sets through recycling views and different layout managers.

### ****CardView in Fragment: Introduction, Life Cycle, Implementation****

#### ****Introduction to CardView in Fragments****

* **CardView** is a layout widget introduced in Android that provides a material design card, which can have rounded corners and shadow effects. It's commonly used to display content in a card-like interface.
* **Fragment** is a part of an activity, allowing for modular design, which can be reused across multiple activities. Combining **CardView** with **Fragment** allows you to display data inside cards within a reusable UI section.
* **CardView** is often used with **RecyclerView** in fragments to display a list of items, each wrapped inside a card.

#### ****Fragment Life Cycle****

A **Fragment** has a distinct life cycle that is closely tied to the **Activity** life cycle. Here's a basic overview:

1. **onAttach()**: Called when the fragment is first attached to its activity.
2. **onCreate()**: Called to initialize the fragment. No UI is created yet.
3. **onCreateView()**: Called to create the view for the fragment. UI components like **CardView** can be initialized here.
4. **onActivityCreated()**: Called after the activity's **onCreate()** method finishes.
5. **onStart()**: Fragment becomes visible but is not interactive yet.
6. **onResume()**: Fragment is now visible and interactive to the user.
7. **onPause()**: Called when the fragment is no longer in the foreground.
8. **onStop()**: Called when the fragment is no longer visible.
9. **onDestroyView()**: Called when the view associated with the fragment is being destroyed.
10. **onDetach()**: Called when the fragment is detached from its activity.

#### ****Implementation of CardView in Fragment****

1. **Add Dependencies**: To use **CardView** in your project, add this line to your build.gradle file:
2. implementation 'androidx.cardview:cardview:1.0.0'
3. **Fragment Layout (XML)**: A **RecyclerView** is used to hold a list of **CardView** items in the fragment's layout.
4. <!-- fragment\_cardview.xml -->
5. <androidx.recyclerview.widget.RecyclerView
6. android:id="@+id/recyclerView"
7. android:layout\_width="match\_parent"
8. android:layout\_height="match\_parent"/>
9. **Item Layout with CardView**: Each item is wrapped inside a **CardView**.
10. <!-- item\_card.xml -->
11. <androidx.cardview.widget.CardView xmlns:app="http://schemas.android.com/apk/res-auto"
12. android:layout\_width="match\_parent"
13. android:layout\_height="wrap\_content"
14. app:cardCornerRadius="8dp"
15. app:cardElevation="4dp">
16. <LinearLayout
17. android:layout\_width="match\_parent"
18. android:layout\_height="wrap\_content"
19. android:orientation="vertical"
20. android:padding="16dp">
21. <TextView
22. android:id="@+id/itemTitle"
23. android:text="Title"
24. android:textSize="18sp"/>
25. <TextView
26. android:id="@+id/itemDescription"
27. android:text="Description"/>
28. </LinearLayout>
29. </androidx.cardview.widget.CardView>
30. **Fragment Code (Java)**: Set up the **RecyclerView** and **CardView** in the fragment, using an adapter to bind data to the cards.
31. public class CardViewFragment extends Fragment {
32. private RecyclerView recyclerView;
33. private MyAdapter adapter;
34. private List<Item> itemList;
35. @Override
36. public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
37. View rootView = inflater.inflate(R.layout.fragment\_cardview, container, false);
38. recyclerView = rootView.findViewById(R.id.recyclerView);
39. recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
40. itemList = new ArrayList<>();
41. itemList.add(new Item("Title 1", "Description 1"));
42. itemList.add(new Item("Title 2", "Description 2"));
43. adapter = new MyAdapter(itemList);
44. recyclerView.setAdapter(adapter);
45. return rootView;
46. }
47. private class MyAdapter extends RecyclerView.Adapter<MyAdapter.MyViewHolder> {
48. private List<Item> items;
49. public MyAdapter(List<Item> items) {
50. this.items = items;
51. }
52. @NonNull
53. @Override
54. public MyViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
55. View itemView = LayoutInflater.from(parent.getContext()).inflate(R.layout.item\_card, parent, false);
56. return new MyViewHolder(itemView);
57. }
58. @Override
59. public void onBindViewHolder(@NonNull MyViewHolder holder, int position) {
60. Item item = items.get(position);
61. holder.title.setText(item.getTitle());
62. holder.description.setText(item.getDescription());
63. }
64. @Override
65. public int getItemCount() {
66. return items.size();
67. }
68. public class MyViewHolder extends RecyclerView.ViewHolder {
69. public TextView title, description;
70. public MyViewHolder(View view) {
71. super(view);
72. title = view.findViewById(R.id.itemTitle);
73. description = view.findViewById(R.id.itemDescription);
74. }
75. }
76. }
77. public class Item {
78. private String title;
79. private String description;
80. public Item(String title, String description) {
81. this.title = title;
82. this.description = description;
83. }
84. public String getTitle() {
85. return title;
86. }
87. public String getDescription() {
88. return description;
89. }
90. }
91. }

#### ****Summary Example****

In this setup, a **Fragment** contains a **RecyclerView** that displays a list of items. Each item is shown in a **CardView** with a title and description. The adapter binds the data to the **CardView** inside the **RecyclerView**.

* **CardView** provides a material design look with rounded corners and shadows.
* **Fragment** manages the UI section that uses **CardView** for displaying a list of items.

This is a simple implementation for teaching purposes to show how **CardView** and **Fragment** work together.