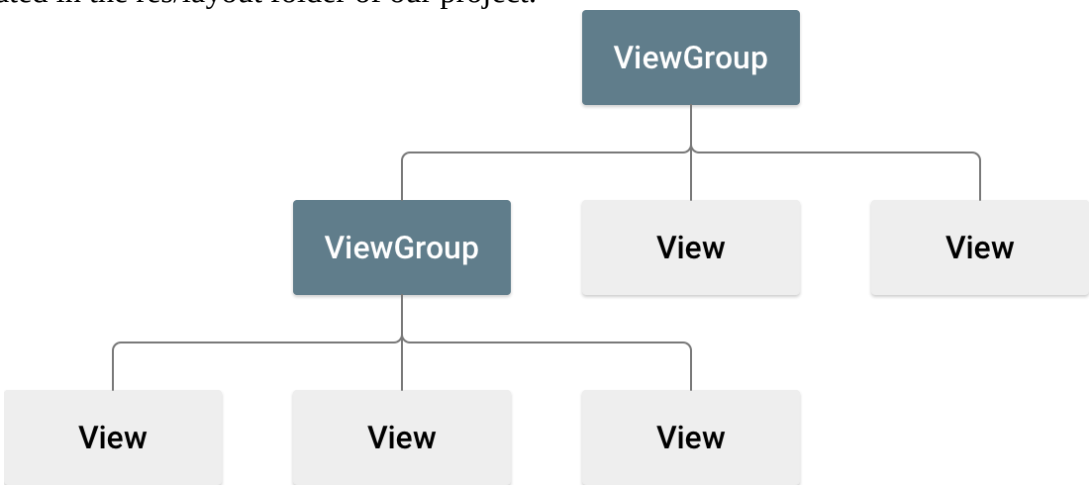
Unit – 3 Android User Interface Design

● UI Overview:

→ The basic building block for user interface is a View object which is created from the View class and occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components like buttons, text fields, etc.

→ The ViewGroup is a subclass of View and provides invisible container that hold other Views or other ViewGroups and define their layout properties.

→ At third level we have different layouts which are subclasses of ViewGroup class and a typical layout defines the visual structure for an Android user interface and can be created either at run time using View/ViewGroup objects or we can declare our layout using simple XML file main_layout.xml which is located in the res/layout folder of our project.



→ All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with. Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects.

→ The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView. The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout .

→ We can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
- **Instantiate layout elements at runtime**. Our app can create View and ViewGroup objects (and manipulate their properties) programmatically.

→ Declaring our UI in XML allows we to separate the presentation of our app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (discussed further in Supporting Different Screen Sizes).

→ The Android framework gives we the flexibility to use either or both of these methods to build our app's UI. For example, we can declare our app's default layouts in XML, and then modify the layout at runtime.

❖ Write the XML

→ Using Android's XML vocabulary, we can quickly design UI layouts and the screen elements they contain, in the same way we create web pages in HTML — with a series of nested elements.

→ Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once we've defined the root element, we can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines our layout.

→ For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

```
<?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:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

→ After we've declared our layout in XML, save the file with the .xml extension, in our Android project's res/layout/ directory, so it will properly compile.
→ More information about the syntax for a layout XML file is available in the Layout Resources document.

❖ Load the XML Resource

→ When we compile our app, each XML layout file is compiled into a View resource. We should load the layout resource from our app code, in our Activity.onCreate() callback implementation. Do so by calling setContentView(), passing it the reference to our layout resource in the form of: R.layout.*layout_file_name*.
→ For example, if our XML layout is saved as main_layout.xml, we would load it for our Activity like so:

```
fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

→ The onCreate() callback method in our Activity is called by the Android framework when our Activity is launched.

● UI screen elements:

- TextView
- EditText
- AutoCompleteTextView
- Button
- ImageButton
- ToggleButton
- RadioButton
- RadioGroup
- ProgressDialog
- Spinner
- Time Picker
- Date Picker
- ListView
- GridView
- Fragment

Fragment

→ A Fragment represents a reusable portion of our app's UI. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. Fragments cannot live on their own--they must be *hosted* by an activity or another fragment. The fragment's view hierarchy becomes part of, or *attaches to*, the host's view hierarchy.

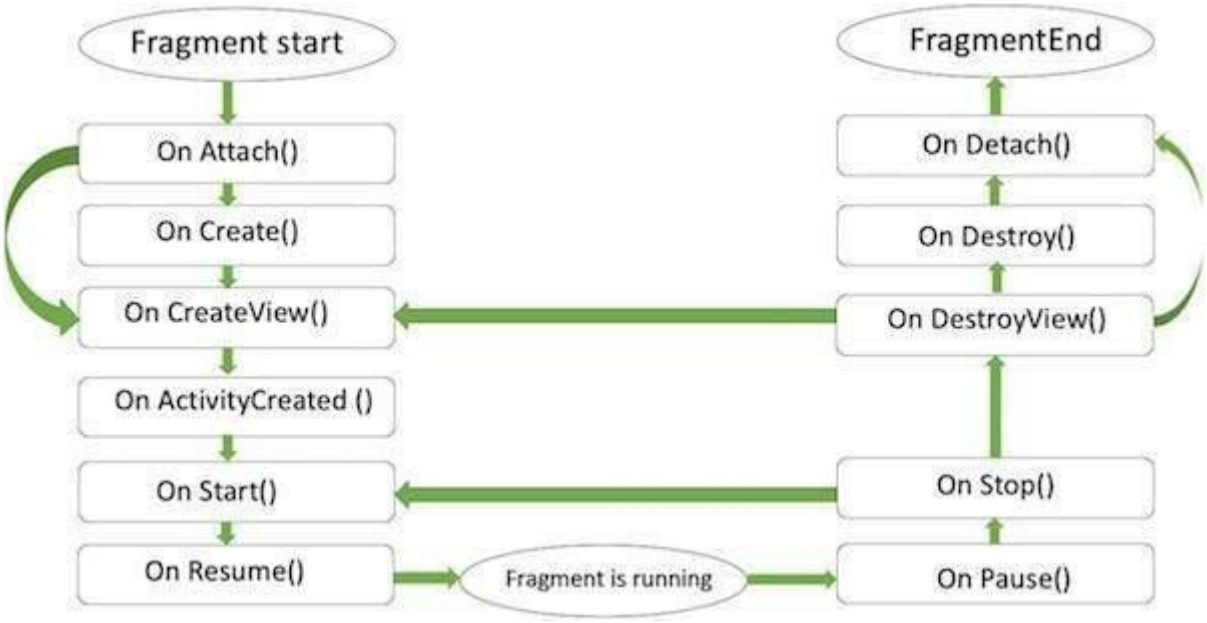
→ Following are important points about fragment −

- A fragment has its own layout and its own behaviour with its own life cycle callbacks.
- We can add or remove fragments in an activity while the activity is running.
- We can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- Fragment life cycle is closely related to the life cycle of its host activity which means when the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behaviour that has no user interface component.
- Fragments were added to the Android API in Honeycomb version of Android which API version 11.

→ We create fragments by extending Fragment class and We can insert a fragment into our activity layout by declaring the fragment in the activity's layout file, as a <fragment> element.

❖ Fragment Life Cycle

→ Android fragments have their own life cycle very similar to an android activity. This section briefs different stages of its life cycle.



→ Here is the list of methods which we can to override in our fragment class −

- **onAttach():** The fragment instance is associated with an activity instance.The fragment and the activity is not fully initialized. Typically we get in this method a reference to the activity which uses the fragment for further initialization work.
- **onCreate():** The system calls this method when creating the fragment. We should initialize essential components of the fragment that we want to retain when the fragment is paused or stopped, then resumed.
- **onCreateView():** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for our fragment, we must return a View component from this method that is the root of our fragment's layout. We can return null if the fragment does not provide a UI.
- **onActivityCreated():** The onActivityCreated() is called after the onCreateView() method when the host activity is created. Activity and fragment instance have been created as well as the view hierarchy of the activity. At this point, view can be accessed with the findViewById() method. example. In this method we can instantiate objects which require a Context object
- **onStart():** The onStart() method is called once the fragment gets visible.

- **onResume():** Fragment becomes active.
- **onPause():** The system calls this method as the first indication that the user is leaving the fragment. This is usually where we should commit any changes that should be persisted beyond the current user session.
- **onStop():** Fragment going to be stopped by calling onStop()
- **onDestroyView():** Fragment view will destroy after call this method
- **onDestroy():** onDestroy() called to do final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

❖ Types of Fragments

→ Basically fragments are divided as three stages as shown below.

- **Single frame fragments** − Single frame fragments are for hand hold devices like mobiles, here we can show only one fragment as a view.
- **List fragments** − fragments having special list view is called as list fragment
- **Fragments transaction** − Using with fragment transaction. we can move one fragment to another fragment.

❖ Single Frame Example:

→ fragment_first.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FirstFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="First Fragment"
        android:gravity="center"
        android:textSize="36dp"
        android:background="@android:color/darker_gray"/>

</FrameLayout>
```

→ FirstFragment.kt

```
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class FirstFragment : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_first, container, false)
```

```
    }
}
```

→ fragment_second.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SecondFragment">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:textSize="36dp"
        android:background="@android:color/holo_blue_light"
        android:text="Second Fragment" />

</FrameLayout>
```

→ SecondFragment.kt

```
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup

class SecondFragment : Fragment() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_second, container, false)
    }
}
```

→ activity_main.xml

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
```

```
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <!-- Button to display first fragment -->
    <Button
        android:id="@+id/button1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="20dp"
        android:layout_marginEnd="20dp"
        android:background="#4CAF50"
        android:onClick="selectFragment"
        android:text="Display Fragment First"
        android:textColor="@android:color/background_light"
        android:textSize="18sp"
        android:textStyle="bold" />

    <!-- Button to display second fragment -->
    <Button
        android:id="@+id/button2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="20dp"
        android:layout_marginTop="20dp"
        android:layout_marginEnd="20dp"
        android:layout_marginBottom="20dp"
        android:background="#4CAF50"
        android:onClick="selectFragment"
        android:text="Display Fragment Second"
        android:textColor="@android:color/background_light"
        android:textSize="18sp"
        android:textStyle="bold" />
    <fragment
        android:id="@+id/fragment_section"
        android:name="com.example.fragmentexample.FirstFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginStart="10dp"
        android:layout_marginEnd="10dp"
        android:layout_marginBottom="10dp"
        tools:layout="@layout/fragment_first" />
</LinearLayout>
```

→ MainActivity.kt

```
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.Fragment
import androidx.fragment.app.FragmentTransaction


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    fun selectFragment(view: View) {
        // creating object for Fragment
        val fr: Fragment
        // displaying first fragment if button1 is clicked
        if (view === findViewById<View>(R.id.button1)) {
            fr = FirstFragment()
        } else {
            fr = SecondFragment()
        }
        // fragment transaction to add or replace
        // fragments while activity is running
        val fragmentTransaction: FragmentTransaction = supportFragmentManager.beginTransaction()
        fragmentTransaction.replace(R.id.fragment_section, fr)

        // making a commit after the transaction
        // to assure that the change is effective
        fragmentTransaction.commit()
    }

}
```

● Layouts:

→ Android **Layout** is used to define the user interface which holds the UI controls or widgets that will appear on the screen of an android application or activity.

→ Generally, every application is combination of View and ViewGroup. As we know, an android application contains a large number of activities and we can say each activity is one page of the application. So, each activities contains multiple user interface components and those components are the instances of the View and ViewGroup.

→ Layout Managers (or simply layouts) are said to be extensions of the ViewGroup class. They are used to set the position of child Views within the UI we are building. We can nest the layouts, and therefore we can create arbitrarily complex UIs using a combination of layouts.

→ There is a number of layout classes in the Android SDK. They can be used, modified or can create our own to make the UI for our Views, Fragments and Activities. We can display our contents effectively by using the right combination of layouts.

❖ Types of Android Layout

1) LinearLayout

→ A LinearLayout aligns each of the child View in either a vertical or a horizontal line. A vertical layout has a column of Views, whereas in a horizontal layout there is a row of Views. It supports a weight attribute for each child View that can control the relative size of each child View within the available space.

→ **LinearLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:baselineAligned	This must be a boolean value, either "true" or "false" and prevents the layout from aligning its children's baselines.
android:baselineAlignedChildIndex	When a linear layout is part of another layout that is baseline aligned, it can specify which of its children to baseline align.
android:divider	This is drawable to use as a vertical divider between buttons. We use a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:orientation	This specifies the direction of arrangement and we will use "horizontal" for a row, "vertical" for a column. The default is horizontal.
android:weightSum	Sum up of child weight

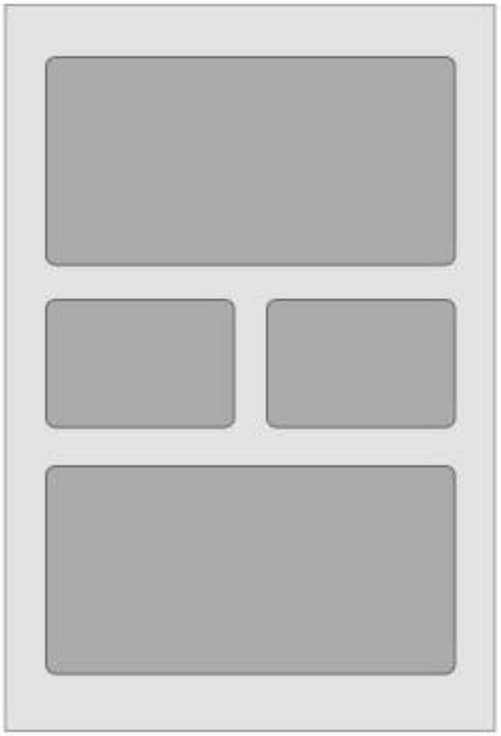
→ Example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical" >

  <Button android:id="@+id/btnStartService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="start_service"/>

  <Button android:id="@+id/btnPauseService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="pause_service"/>

  <Button android:id="@+id/btnStopService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="stop_service"/>
```

```
</LinearLayout>
```

2) RelativeLayout

→ It is flexible than other native layouts as it lets us to define the position of each child View relative to the other views and the dimensions of the screen.



→ **RelativeLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:ignoreGravity	This indicates what view should not be affected by gravity.

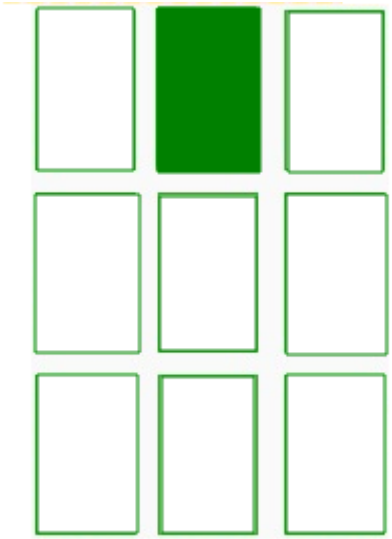
Example:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:paddingLeft="16dp"
  android:paddingRight="16dp" >

  <EditText
    android:id="@+id/name"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/reminder" />

  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_alignParentStart="true"
```

```
    android:layout_below="@+id/name">

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="New Button"
      android:id="@+id/button" />

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="New Button"
      android:id="@+id/button2" />

  </LinearLayout>

</RelativeLayout>
```

3) FrameLayout

→ It is the simplest of the Layout Managers that pins each child view within its frame. By default the position is the top-left corner, though the gravity attribute can be used to alter its locations. We can add multiple children stacks each new child on top of the one before, with each new View potentially obscuring the previous ones.



→ **FrameLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:foreground	This defines the drawable to draw over the content and possible values may be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:foregroundGravity	Defines the gravity to apply to the foreground drawable. The gravity defaults to fill. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:measureAllChildren	Determines whether to measure all children or just those in the VISIBLE or INVISIBLE state when measuring. Defaults to false.

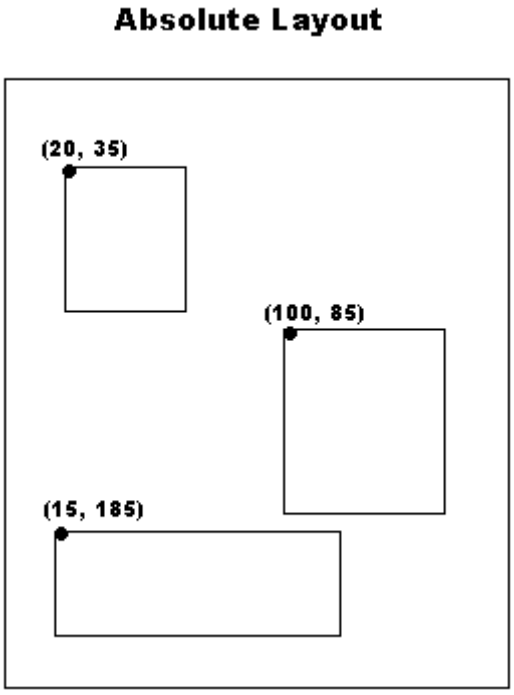
→ **Example:**

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
```

```
  <ImageView
    android:src="@drawable/ic_launcher"
    android:scaleType="fitCenter"
    android:layout_height="250px"
    android:layout_width="250px"/>

  <TextView
    android:text="Frame Demo"
    android:textSize="30px"
    android:textStyle="bold"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:gravity="center"/>
</FrameLayout>
```

4) Absolute Layout

→ An Absolute Layout lets we specify exact locations (x/y coordinates) of its children. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.



→ **AbsoluteLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:layout_x	This specifies the x-coordinate of the view.
android:layout_y	This specifies the y-coordinate of the view.

→ Example:

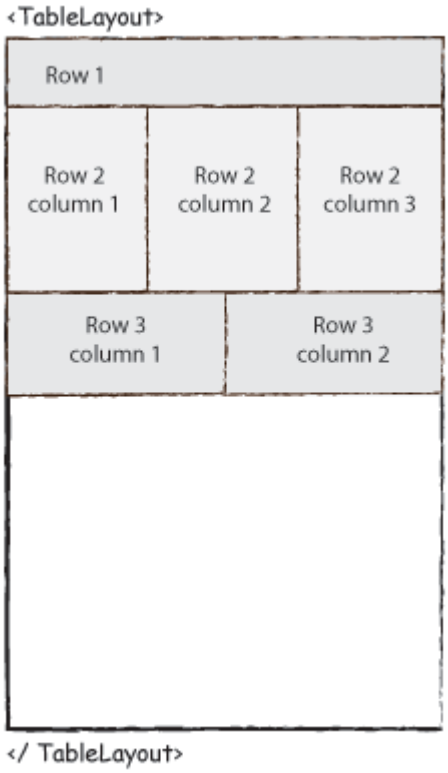
```
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <Button
    android:layout_width="100dp"
```

```
    android:layout_height="wrap_content"
    android:text="OK"
    android:layout_x="50px"
    android:layout_y="361px" />
  <Button
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:text="Cancel"
    android:layout_x="225px"
    android:layout_y="361px" />

</AbsoluteLayout>
```

5) Table Layout

→ Android TableLayout going to be arranged groups of views into rows and columns. We will use the <TableRow> element to build a row in the table. Each row has zero or more cells; each cell can hold one View object.

→ TableLayout containers do not display border lines for their rows, columns, or cells.



→ **TableLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:collapseColumns	This specifies the zero-based index of the columns to collapse. The column indices must be separated by a comma: 1, 2, 5.
android:shrinkColumns	The zero-based index of the columns to shrink. The column indices must be separated by a comma: 1, 2, 5.
android:stretchColumns	The zero-based index of the columns to stretch. The column indices must be separated by a comma: 1, 2, 5.

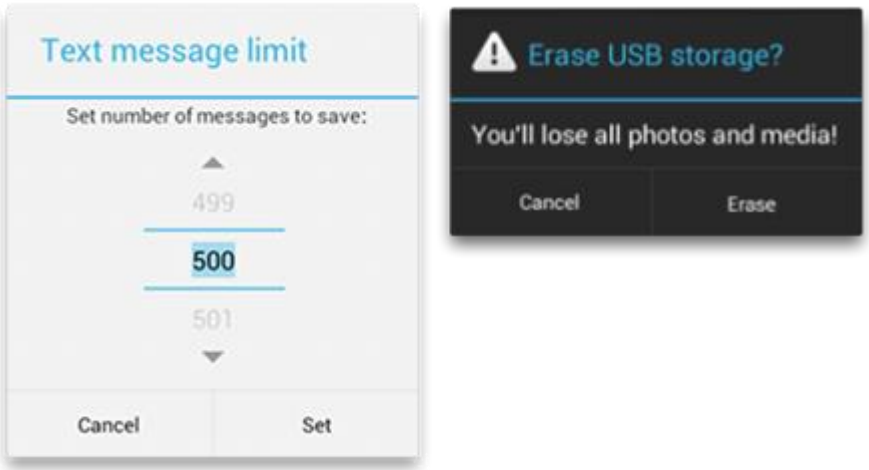
→ Example:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
      android:text="Time"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <TextClock
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/textClock"
      android:layout_column="2" />

  </TableRow>

  <TableRow>

    <TextView
      android:text="First Name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <EditText
      android:width="200px"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
  </TableRow>

  <TableRow>

    <TextView
      android:text="Last Name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <EditText
      android:width="100px"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
  </TableRow>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
```

```
    <RatingBar
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/ratingBar"
      android:layout_column="2" />
  </TableRow>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"/>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Submit"
      android:id="@+id/button"
      android:layout_column="2" />
  </TableRow>

</TableLayout>
```

● Dialogs:

→ A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.



→ The Dialog class is the base class for dialogs, but we should avoid instantiating Dialog directly. Instead, use one of the following subclasses:

❖ AlertDialog

→ Android AlertDialog can be used to display the dialog message with OK and Cancel buttons. It can be used to interrupt and ask the user about his/her choice to continue or discontinue.

→ The AlertDialog class allows we to build a variety of dialog designs and is often the only dialog class we'll need. there are three regions of an alert dialog: Android AlertDialog is the subclass of Dialog class.

i. Title

→ This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If we need to state a simple message or question (such as the dialog in figure 1), we don't need a title.

ii. Content area

→ This can display a message, a list, or other custom layout.

iii. Action buttons

→ There should be no more than three action buttons in a dialog.
→ The AlertDialog.Builder class provides APIs that allow we to create an AlertDialog with these kinds of content, including a custom layout.

Method	Description
public AlertDialog.Builder setTitle(CharSequence)	This method is used to set the title of AlertDialog.
public AlertDialog.Builder setMessage(CharSequence)	This method is used to set the message for AlertDialog.
public AlertDialog.Builder setIcon(int)	This method is used to set the icon over AlertDialog.

Adding buttons

→ To add action buttons we have to call the setPositiveButton() and setNegativeButton() methods:
→ The set...Button() methods require a title for the button (supplied by a string resource) and a DialogInterface.OnClickListener that defines the action to take when the user presses the button.
→ There are three different action buttons we can add:
- **Positive**
  - We should use this to accept and continue with the action (the "OK" action).
- **Negative**
  - We should use this to cancel the action.
- **Neutral**
  - We should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

Adding a list

→ There are three kinds of lists available with the AlertDialog APIs:
- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

❖ DatePickerDialog or TimePickerDialog

→ A dialog with a pre-defined UI that allows the user to select a date or time.
→ DatePickerDialog & TimePickerDialog classes have onDateSetListener() & onTimeSetListener() callback methods respectively.
→ These callback methods are invoked when the user is done with filling the date and time respectively.
→ The DatePickerDialog class consists of a 5 argument constructor with the parameters listed below.
- **Context**: It requires the application context
- **CallBack Function**: onDateSet() is invoked when the user sets the date with the following parameters:
  - int year : It will be store the current selected year from the dialog
  - int monthOfYear : It will be store the current selected month from the dialog

        - int dayOfMonth : It will be store the current selected day from the dialog
    - **int mYear** : It shows the the current year that's visible when the dialog pops up
    - **int mMonth** : It shows the the current month that's visible when the dialog pops up
    - **int mDay** : It shows the the current day that's visible when the dialog pops up

→ The TimePickerDialog class consists of a 5 argument constructor with the parameters listed below.

- **Context**: It requires the application context
- **CallBack Function**: is invoked when the user sets the time with the following parameters:
    - int hourOfDay : It will be store the current selected hour of the day from the dialog
    - int minute : It will be store the current selected minute from the dialog
- **int mHours** : It shows the the current Hour that's visible when the dialog pops up
- **int mMinute** : It shows the the current minute that's visible when the dialog pops up
- **boolean false** : If its set to false it will show the time in 24 hour format else not

→ **Example:**

→ **AlertDialog Example**

➢ **activity_main.xml**

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

   <Button
      android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginStart="8dp"
      android:layout_marginTop="8dp"
      android:text="Click Here"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent" />

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

➢ **Main_Activity.kt**

```
import android.app.AlertDialog
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.Toast

class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
      val button = findViewById<Button>(R.id.button)
```

```
        button.setOnClickListener {
            val builder = AlertDialog.Builder(this)
            //set title for alert dialog
            builder.setTitle("Delete File")
            //set message for alert dialog
            builder.setMessage("Deleting file may be harm your system")
            builder.setIcon(android.R.drawable.ic_dialog_alert)

            //performing positive action
            builder.setPositiveButton("Yes"){dialogInterface, which ->
                Toast.makeText(applicationContext,"clicked yes",Toast.LENGTH_LONG).show()
            }
            //performing cancel action
            builder.setNeutralButton("Cancel"){dialogInterface , which ->
                Toast.makeText(applicationContext,"clicked cancel\n operation
cancel",Toast.LENGTH_LONG).show()
            }
            //performing negative action
            builder.setNegativeButton("No"){dialogInterface, which ->
                Toast.makeText(applicationContext,"clicked No",Toast.LENGTH_LONG).show()
            }
            // Create the AlertDialog
            val alertDialog: AlertDialog = builder.create()
            // Set other dialog properties
            alertDialog.setCancelable(false)
            alertDialog.show()
        }
    }
}
```

→ **DatePickerDialog Example:**

- **activity_main.xml**

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

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Hello World!"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="Open Date Picker"
        android:onClick="clickDataPicker"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />

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

- **Main_Activity.kt**

```
import android.app.AlertDialog
import android.app.DatePickerDialog
import android.os.Build
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.Button
import android.widget.Toast
import androidx.annotation.RequiresApi
import java.util.*

class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)
   }

   @RequiresApi(Build.VERSION_CODES.N)
   fun clickDataPicker(view: View) {
      val c = Calendar.getInstance()
      val year = c.get(Calendar.YEAR)
      val month = c.get(Calendar.MONTH)
      val day = c.get(Calendar.DAY_OF_MONTH)

      val dpd = DatePickerDialog(this, DatePickerDialog.OnDateSetListener { view, year,
monthOfYear, dayOfMonth ->
         // Display Selected date in Toast
         Toast.makeText(this, """$dayOfMonth - ${monthOfYear + 1} - $year""",
Toast.LENGTH_LONG).show()

      }, year, month, day)
      dpd.show()
   }
}
```

● Animation:

→ Animations can add visual cues that notify users about what's going on in our app. They are especially useful when the UI changes state, such as when new content loads or new actions become available. Animations also add a polished look to our app, which gives it a higher quality look and feel.

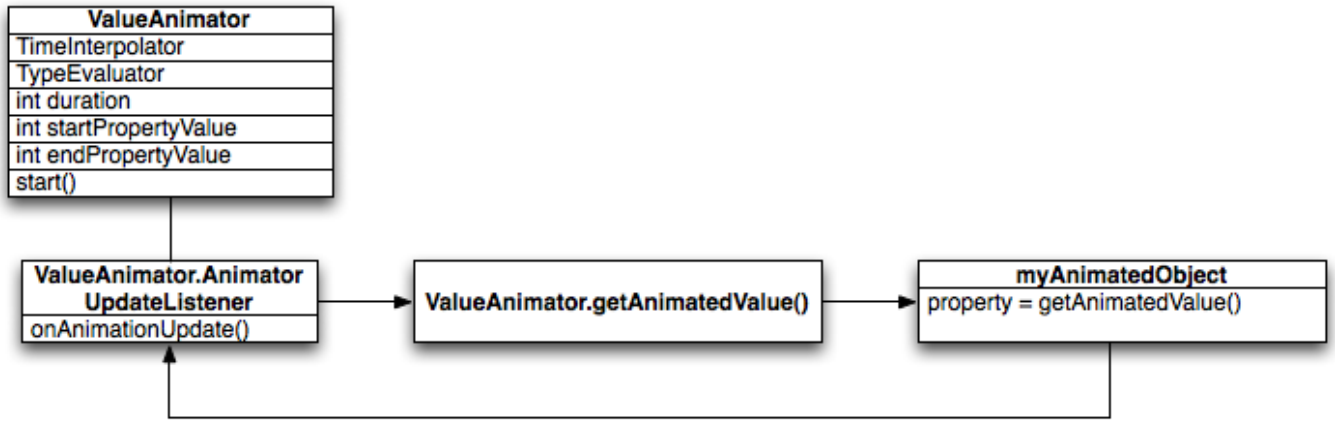
→ Android includes different animation APIs depending on what type of animation we want, so this page provides an overview of the different ways we can add motion to our UI.

1) Property Animation

→ The property animation system is a robust framework that allows us to animate almost anything. We can define an animation to change any object property over time, regardless of whether it draws to the screen or not. A property animation changes a property's (a field in an object) value over a specified length of time. To animate something, we specify the object property that we want to animate, such as an object's position on the screen, how long we want to animate it for, and what values we want to animate between.

→ Characteristics of an animation:

- **Duration:** We can specify the duration of an animation. The default length is 300 ms.
- **Time interpolation:** We can specify how the values for the property are calculated as a function of the animation's current elapsed time.
- **Repeat count and behavior:** We can specify whether or not to have an animation repeat when it reaches the end of a duration and how many times to repeat the animation. We can also specify whether we want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- **Animator sets:** We can group animations into logical sets that play together or sequentially or after specified delays.
- **Frame refresh delay:** We can specify how often to refresh frames of our animation. The default is set to refresh every 10 ms, but the speed in which our application can refresh frames is ultimately dependent on how busy the system is overall and how fast the system can service the underlying timer.



→ file location:
res/animator/*filename*.xml
The filename will be used as the resource ID.

→ compiled resource datatype:
Resource pointer to a ValueAnimator, ObjectAnimator, or AnimatorSet.

→ resource reference:
In Kotlin: R.animator.*filename*
In XML: @[*package*:]animator/*filename*

→ syntax:

```
<set
  android:ordering=["together" | "sequentially"]>

    <objectAnimator
      android:propertyName="string"
      android:duration="int"
      android:valueFrom="float | int | color"
      android:valueTo="float | int | color"
```

```
        android:startOffset="int"
        android:repeatCount="int"
        android:repeatMode=["repeat" | "reverse"]
        android:valueType=["intType" | "floatType"]/>

    <animator
        android:duration="int"
        android:valueFrom="float | int | color"
        android:valueTo="float | int | color"
        android:startOffset="int"
        android:repeatCount="int"
        android:repeatMode=["repeat" | "reverse"]
        android:valueType=["intType" | "floatType"]/>

    <set>
        ...
    </set>
</set>
```

→ The file must have a single root element: either <set>, <objectAnimator>, or <valueAnimator>. We can group animation elements together inside the <set> element, including other <set> elements.

→ Elements:

1. <set>

⇒ A container that holds other animation elements (<objectAnimator>, <valueAnimator>, or other <set> elements). Represents an AnimatorSet.

⇒ We can specify nested <set> tags to further group animations together. Each <set> can define its own ordering attribute.

⇒ **Attributes:**

- **android:ordering**
  Keyword. Specifies the play ordering of animations in this set.

Value	Description
sequentially	Play animations in this set sequentially
together (default)	Play animations in this set at the same time.

2. <objectAnimator>

⇒ Animates a specific property of an object over a specific amount of time. Represents an ObjectAnimator.

⇒ **Attributes:**

- **android:propertyName**
  String. ***Required***. *The object's property to animate, referenced by its name. For example we can specify "alpha" or "backgroundColor" for a View object.*
  The objectAnimator element does not expose a target attribute, however, so we cannot set the object to animate in the XML declaration. We have to inflate our animation XML resource by calling loadAnimator() and call setTarget() to set the target object that contains this property.
- **android:valueTo**
  float, int, or color. ***Required***. *The value where the animated property ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:valueFrom**
  float, int, or color. *The value where the animated property starts. If not specified, the animation starts at the value obtained by the property's get method. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:duration**

int. *The time in milliseconds of the animation. 300 milliseconds is the default.*

- **android:startOffset**
  int. *The amount of milliseconds the animation delays after start() is called.*
- **android:repeatCount**
  int. *How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.*
- **android:repeatMode**
  int. *How an animation behaves when it reaches the end of the animation. android:repeatCount must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.*
- **android:valueType**
  Keyword. *Do not specify this attribute if the value is a color. The animation framework automatically handles color values*

Value	Description
intType	Specifies that the animated values are integers
floatType (default)	Specifies that the animated values are floats

3. <animator>

⇒ Animates an over a specified amount of time. Represents a ValueAnimator.

⇒ **Attributes:**

- **android:valueTo**
  float, int, or color. ***Required***. *The value where the animation ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:valueFrom**
  float, int, or color. ***Required***. *The value where the animation starts. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:duration**
  int. *The time in milliseconds of the animation. 300ms is the default.*
- **android:startOffset**
  int. *The amount of milliseconds the animation delays after start() is called.*
- **android:repeatCount**
  int. *How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.*
- **android:repeatMode**
  int. *How an animation behaves when it reaches the end of the animation. android:repeatCount must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.*
- **android:valueType**
  Keyword. *Do not specify this attribute if the value is a color. The animation framework automatically handles color values.*

Value	Description
intType	Specifies that the animated values are integers
floatType (default)	Specifies that the animated values are floats

→ The Animator class provides the basic structure for creating animations. We normally do not use this class directly as it only provides minimal functionality that must be extended to fully support animating values. The following subclasses extend Animator:

❖ ValueAnimator

→ This class provides a simple timing engine for running animations which calculate animated values and set them on target objects.
→ There is a single timing pulse that all animations use. It runs in a custom handler to ensure that property changes happen on the UI thread.
→ By default, ValueAnimator uses non-linear time interpolation, via the AccelerateDecelerateInterpolator class, which accelerates into and decelerates out of an animation. This behavior can be changed by calling ValueAnimator#setInterpolator(TimeInterpolator).
→ Animators can be created from either code or resource files. Here is an example of a ValueAnimator resource file:

```
<animator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueFrom="1"
    android:valueTo="0"
    android:valueType="floatType"
    android:repeatCount="1"
    android:repeatMode="reverse"/>
```

→ **Methods:**

open Any! getAnimatedValue(propertyName: String!)	The most recent value calculated by this ValueAnimator for propertyName.
open Long getCurrentPlayTime()	Gets the current position of the animation in time, which is equal to the current time minus the time that the animation started.
open Long getDuration()	Gets the length of the animation.
open Int getRepeatCount()	Defines how many times the animation should repeat.
open Int getRepeatMode()	Defines what this animation should do when it reaches the end.
open Long getStartDelay()	The amount of time, in milliseconds, to delay starting the animation after start() is called.
open Long getTotalDuration()	Gets the total duration of the animation, accounting for animation sequences, start delay, and repeating.
open Array<PropertyValuesHolder!>! getValues()	Returns the values that this ValueAnimator animates between.
open Boolean isRunning()	Returns whether this Animator is currently running (having been started and gone past any initial startDelay period and not yet ended).
open Boolean isStarted()	Returns whether this Animator has been started and not yet ended.
open Unit start()	Starts this animation.
open Unit pause()	Pauses a running animation.
open Unit resume()	Resumes a paused animation, causing the animator to pick up where it left off when it was paused.
open Unit reverse()	Plays the ValueAnimator in reverse.
open ValueAnimator! setDuration(duration: Long)	Sets the length of the animation.
open Unit setRepeatCount(value: Int)	Sets how many times the animation should be repeated.
open Unit setRepeatMode(value: Int)	Defines what this animation should do when it reaches the end.

❖ ObjectAnimator

→ This subclass of ValueAnimator provides support for animating properties on target objects. The constructors of this class take parameters to define the target object that will be animated as well as the

name of the property that will be animated. Appropriate set/get functions are then determined internally and the animation will call these functions as necessary to animate the property.

→ Animators can be created from either code or resource files, as shown here:

```
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueTo="200"
    android:valueType="floatType"
    android:propertyName="y"
    android:repeatCount="1"
    android:repeatMode="reverse"/>
```

→ **Methods of ObjectAnimator Class:**

String? getPropertyName()	Gets the name of the property that will be animated.
Any? getTarget()	The target object whose property will be animated by this animation
ObjectAnimator setDuration(duration: Long)	Sets the length of the animation.
Unit setFloatValues(vararg values: Float)	Sets float values that will be animated between.
Unit setIntValues(vararg values: Int)	Sets int values that will be animated between.
Unit setProperty(property: Property<Any!, Any!>)	Sets the property that will be animated.
Unit setPropertyName(propertyName: String)	Sets the name of the property that will be animated.
Unit start ()	Starts this animation.

❖ AnimatorSet

→ This class plays a set of Animator objects in the specified order. Animations can be set up to play together, in sequence, or after a specified delay.

→ There are two different approaches to adding animations to a AnimatorSet: either the playTogether() or playSequentially() methods can be called to add a set of animations all at once, or the AnimatorSet#play(Animator) can be used in conjunction with methods in the Builder class to add animations one by one.

→ It is possible to set up a AnimatorSet with circular dependencies between its animations. For example, an animation a1 could be set up to start before animation a2, a2 before a3, and a3 before a1. The results of this configuration are undefined, but will typically result in none of the affected animations being played. Because of this (and because circular dependencies do not make logical sense anyway), circular dependencies should be avoided, and the dependency flow of animations should only be in one direction.

→ In many cases, we want to play an animation that depends on when another animation starts or finishes. The Android system lets us bundle animations together into an AnimatorSet, so that we can specify whether to start animations simultaneously, sequentially, or after a specified delay. We can also nest AnimatorSet objects within each other.

→ The following code snippet plays the following Animator objects in the following manner:

- Plays bounceAnim.
- Plays squashAnim1, squashAnim2, stretchAnim1, and stretchAnim2 at the same time.
- Plays bounceBackAnim.
- Plays fadeAnim.

→ **Example:**

```
val bouncer = AnimatorSet().apply {
    play(bounceAnim).before(squashAnim1)
    play(squashAnim1).with(squashAnim2)
    play(squashAnim1).with(stretchAnim1)
    play(squashAnim1).with(stretchAnim2)
    play(bounceBackAnim).after(stretchAnim2)
}
val fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f).apply {
    duration = 250
```

```
}
AnimatorSet().apply {
   play(bouncer).before(fadeAnim)
   start()
}
```

→ **Methods**

Long getDuration ()	Gets the length of each of the child animations of this AnimatorSet.
AnimatorSet! setDuration (duration: Long)	Sets the length of each of the current child animations of this AnimatorSet.
Long getCurrentPlayTime ()	Returns the milliseconds elapsed since the start of the animation.
Unit setCurrentPlayTime (playtime: Long)	Sets the position of the animation to the specified point in time.
Long getStartDelay ()	The amount of time, in milliseconds, to delay starting the animation after start() is called.
Unit setStartDelay (startDelay: Long)	The amount of time, in milliseconds, to delay starting the animation after start() is called.
Long getTotalDuration ()	Gets the total duration of the animation, accounting for animation sequences, start delay, and repeating. Return DURATION_INFINITE if the duration is infinite.
Boolean isRunning ()	Returns true if any of the child animations of this AnimatorSet have been started and have not yet ended.
Boolean isStarted ()	Returns whether this Animator has been started and not yet ended.
Unit pause ()	Pauses a running animation.
AnimatorSet.Builder! play (anim: Animator!)	This method creates a Builder object, which is used to set up playing constraints.
Unit playSequentially (vararg items: Animator!)	Sets up this AnimatorSet to play each of the supplied animations when the previous animation ends.
Unit playTogether (vararg items: Animator!)	Sets up this AnimatorSet to play all of the supplied animations at the same time.
Unit resume ()	Resumes a paused animation, causing the animator to pick up where it left off when it was paused.
Unit reverse ()	Plays the AnimatorSet in reverse.

2) View Animation

→ We can use the view animation system to perform tweened animation on Views. Tween animation calculates the animation with information such as the start point, end point, size, rotation, and other common aspects of an animation.

→ The view animation framework supports both tween and frame by frame animations, which can both be declared in XML. The following sections describe how to use both methods.

1) Tween animation: Creates an animation by performing a series of transformations on a single image with an Animation
2) Frame animation: or creates an animation by showing a sequence of images in order with an AnimationDrawable.

❖ Tween animation

→ An animation defined in XML that performs transitions such as rotating, fading, moving, and stretching on a graphic.

→ file location:

res/anim/*filename*.xml

The filename will be used as the resource ID.

→ compiled resource datatype:
Resource pointer to an Animation.
→ resource reference:
In Kotlin: R.anim.*filename*
In XML: @[*package*:]anim/*filename*
→ **Syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@[package:]anim/interpolator_resource"
  android:shareInterpolator=["true" | "false"] >
  <alpha
    android:fromAlpha="float"
    android:toAlpha="float" />
  <scale
    android:fromXScale="float"
    android:toXScale="float"
    android:fromYScale="float"
    android:toYScale="float"
    android:pivotX="float"
    android:pivotY="float" />
  <translate
    android:fromXDelta="float"
    android:toXDelta="float"
    android:fromYDelta="float"
   android:toYDelta="float" />
  <rotate
    android:fromDegrees="float"
    android:toDegrees="float"
    android:pivotX="float"
    android:pivotY="float" />
  <set>
    ...
  </set>
</set>
```

→ The file must have a single root element: either an <alpha>, <scale>, <translate>, <rotate>, or <set> element that holds a group (or groups) of other animation elements (even nested <set> elements).

→ Elements:

1. <set>

⇒ A container that holds other animation elements (<alpha>, <scale>, <translate>, <rotate>) or other <set> elements. Represents an AnimationSet.
⇒ **Attributes:**
- **android:interpolator**
  Interpolator resource. *An Interpolator to apply on the animation. The value must be a reference to a resource that specifies an interpolator (not an interpolator class name). There are default interpolator resources available from the platform or we can create our own interpolator resource.*
- **android:shareInterpolator**
  Boolean. *"true" if we want to share the same interpolator among all child elements.*

2. <alpha>

⇒ A fade-in or fade-out animation. Represents an AlphaAnimation.
⇒ **Attributes:**
- **android:fromAlpha**

Float. *Starting opacity offset, where 0.0 is transparent and 1.0 is opaque.*

- **android:toAlpha**
  Float. *Ending opacity offset, where 0.0 is transparent and 1.0 is opaque.*

3. <scale>

⇒ A resizing animation. We can specify the center point of the image from which it grows outward (or inward) by specifying pivotX and pivotY. For example, if these values are 0, 0 (top-left corner), all growth will be down and to the right. Represents a ScaleAnimation.

⇒ **Attributes:**

- **android:fromXScale**
  Float. *Starting X size offset, where 1.0 is no change.*
- **android:toXScale**
  Float. *Ending X size offset, where 1.0 is no change.*
- **android:fromYScale**
  Float. *Starting Y size offset, where 1.0 is no change.*
- **android:toYScale**
  Float. *Ending Y size offset, where 1.0 is no change.*
- **android:pivotX**
  Float. *The X coordinate to remain fixed when the object is scaled.*
- **android:pivotY**
  Float. *The Y coordinate to remain fixed when the object is scaled.*

4. <translate>

⇒ A vertical and/or horizontal motion. Supports the following attributes in any of the following three formats: values from -100 to 100 ending with "%", indicating a percentage relative to itself; values from -100 to 100 ending in "%p", indicating a percentage relative to its parent; a float value with no suffix, indicating an absolute value. Represents a TranslateAnimation.

⇒ **Attributes:**

- **android:fromXDelta**
  Float or percentage. Starting X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").
- **android:toXDelta**
  Float or percentage. *Ending X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").*
- **android:fromYDelta**
  Float or percentage. *Starting Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").*
- **android:toYDelta**
  Float or percentage. *Ending Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").*

5. <rotate>

⇒ A rotation animation. Represents a RotateAnimation.

⇒ **Attributes:**

- **android:fromDegrees**
  Float. *Starting angular position, in degrees.*
- **android:toDegrees**
  Float. *Ending angular position, in degrees.*
- **android:pivotX**
  Float or percentage. *The X coordinate of the center of rotation. Expressed either: in pixels relative to the object's left edge (such as "5"), in percentage relative to the object's*

left edge (such as "5%"), or in percentage relative to the parent container's left edge (such as "5%p").

- **android:pivotY**
  Float or percentage. *The Y coordinate of the center of rotation. Expressed either: in pixels relative to the object's top edge (such as "5"), in percentage relative to the object's top edge (such as "5%"), or in percentage relative to the parent container's top edge (such as "5%p").*

→ **Example:**
XML file saved at res/anim/hyperspace_jump.xml:

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="false">
  <scale
    android:interpolator="@android:anim/accelerate_decelerate_interpolator"
    android:fromXScale="1.0"
    android:toXScale="1.4"
    android:fromYScale="1.0"
    android:toYScale="0.6"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fillAfter="false"
    android:duration="700" />
  <set
    android:interpolator="@android:anim/accelerate_interpolator"
    android:startOffset="700">
    <scale
      android:fromXScale="1.4"
      android:toXScale="0.0"
      android:fromYScale="0.6"
      android:toYScale="0.0"
      android:pivotX="50%"
      android:pivotY="50%"
      android:duration="400" />
    <rotate
      android:fromDegrees="0"
      android:toDegrees="-45"
      android:toYScale="0.0"
      android:pivotX="50%"
      android:pivotY="50%"
      android:duration="400" />
  </set>
</set>
```

→ This application code will apply the animation to an ImageView and start the animation:

```
val image = findViewById<ImageView>(R.id.image)
val spacejump = AnimationUtils.loadAnimation(this, R.anim.hyperspace_jump)
image.startAnimation(spacejump)
```

❖ Frame animation

→ An animation defined in XML that shows a sequence of images in order (like a film).
→ file location:
res/drawable/*filename*.xml
The filename will be used as the resource ID.
→ compiled resource datatype:
Resource pointer to an AnimationDrawable.
→ resource reference:

In Kotlin: R.drawable.*filename*
In XML: @[*package*:]drawable.*filename*

→ **Syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot=["true" | "false"] >
  <item
    android:drawable="@[package:]drawable/drawable_resource_name"
    android:duration="integer" />
</animation-list>
```

→ **Elements:**

1. <animation-list>

⇒ **Required**. This must be the root element. Contains one or more <item> elements.

⇒ **Attributes:**

- **android:oneshot**
  Boolean. "true" if we want to perform the animation once; "false" to loop the animation.

2. <item>

⇒ A single frame of animation. Must be a child of a <animation-list> element.

⇒ **Attributes:**

- **android:drawable**
  Drawable resource. The drawable to use for this frame.
- **android:duration**
  Integer. The duration to show this frame, in milliseconds.

→ **Example:**

XML file saved at res/anim/myanim.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
  <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
  <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
  <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

This application code will set the animation as the background for a View, then play the animation:

```
import android.graphics.drawable.AnimationDrawable
import android.os.Bundle
import android.view.View
import android.widget.ImageView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val img: ImageView = findViewById<View>(R.id.iv1) as ImageView
    img.setBackgroundResource(R.drawable.myanim)
    val frameAnimation = img.background as AnimationDrawable
    frameAnimation.start()
  }
}
```